AFRL-IF-RS-TR-2004-111
Final Technical Report
April 2004

AUTOMATED DYNAMIC ASSEMBLY OF DEPENDABLE SYSTEM ARCHITECTURES

SRI International

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K509

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** APRIL 2004	**3. REPORT TYPE AND DATES COVERED** Final Jul 00 – May 03

4. TITLE AND SUBTITLE AUTOMATED DYNAMIC ASSEMBLY OF DEPENDABLE SYSTEM ARCHITECTURES	**5. FUNDING NUMBERS** C - F30602-00-C-0199 PE - 62302E PR - DASA TA - 00 WU - 08
6. AUTHOR(S) R. A. Riemenschneider	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) SRI International 333 Ravenswood Avenue Menlo Park California 94025	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington Virginia 22203-1714; AFRL/IFTB 525 Brooks Road Rome New York 13441-4505	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2004-111

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Deborah A. Cerino/IFTB/(315) 330-1445/ Deborah.Cerino@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 Words)*

This project was part of the DARPA Dynamic Assembly for Systems Adaptability, Dependability, and Assurance (DASADA) Program. The DASADA program objective was to develop dynamic gauges or measures of component composability or interoperability. As military software systems become more complex, it is evident that they must be able to change themselves by swapping or modifying components, changing component interaction protocols, or changing topology. Particularly for critical military systems, we need to enable changes to be made predictably to ensure safety and reliability. The objective of this project was to develop technology for generating custom dependability gauges which monitor dependability properties (e.g., security, safety, fault tolerance) of a complex, evolving software architecture at runtime. This project's principal innovation consists of focusing on abstractions rather than refinement, and on automatic updating of abstractions and analyses developed at design time after making small well-structured changes to architectural requirements and the system architecture. Their dependability gauge technology is complementary to the more fine-grained runtime analysis that can be performed by monitoring events at component interfaces and within connectors that are being developed by other DASADA contractors.

14. SUBJECT TERMS Dependability Properties, Architectural Models	**15. NUMBER OF PAGES** 77
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Objectives

The objective of the project was to develop technology for generating custom *dependability gauges*, which monitor dependability properties (e.g., security, safety, fault tolerance) of a complex, evolving software architecture at runtime.

The principal application of the technology was to demonstrate a re-engineered, dynamically adaptable prototype upgrade of the Structured Evidential Argumentation System (SEAS)[1] developed by the SRI Artificial Intelligence Center (AIC) for DARPA's Project Genoa.

1.2 Approach

The approach was based on previous SRI research on guaranteeing dependability properties of software architectures. In that research, we proved that an abstract description of a software architecture guarantees the desired dependability property and that dependability is preserved by each refinement step in the design process that produces the implementation of the abstract description. Continued dependability as an architecture evolves at runtime was guaranteed by verifying (at design time) that allowed evolutionary transformations preserve dependability.

The principal focus of our DASADA work was to support less-constrained architectural evolution at runtime. When a change is made to the architecture at runtime, the change will be automatically propagated up the chain of refine-

[1]See the Web site `http://www.ai.sri.com/~seas/` for more details about SEAS.

ments leading to the abstract, analyzable description. The dependability level of the abstract description will be automatically reassessed, and then dependability preservation of the refinement steps will be reassessed. The highest demonstrable level of dependability will be displayed on a dependability gauge.

If an evolutionary step reduces dependability to an unacceptable level, a library of dependability-enhancing evolutionary transformations is available that can be applied, in some cases, to restore dependability.

This technology has been demonstrated by developing dependability gauges and a library of dependability-enhancing transformations for the SEAS component of the Genoa system.

1.3 Demonstration Development and Experimentation

In making our intrusion-tolerant version of SEAS more dynamic, we added four intrusion-tolerance mechanisms that can be independently toggled:

1. redundancy in persistent storage
2. encryption of traffic between the persistent store(s) and the main application
3. encryption of the information in the persistent store
4. use of the SSL protocol rather than HTTP between the user and the application

For each of the four mechanisms, a corresponding gauge that measures the effect of toggling the mechanism has been implemented. For example, the gauge that corresponds to storage redundancy measures the length of the shortest attack sequence that compromises storage, while the storage traffic encryption gauge measures whether it is possible for an unauthorized agent (one who lacks the decryption key) to obtain information from the message traffic. Each of the four gauges analyzes a different abstract model, generated from the common concrete architectural model by applying a different abstraction strategy, and uses a different technology to perform the analysis. For example, the storage redundancy gauge generates Promela code from its abstract model, which is then analyzed by the Spin model checker,[2] while the storage traffic encryption gauge generates a logical theory and a query from its abstract model, which is then fed to the PTTP automatic theorem prover[3] to determine whether the query is a logical consequence of the theory.

The intrusion-tolerance mechanisms have been designed to allow two experimental hypotheses to be tested.

1. Instrumenting a system with dependability gauges will improve the assessment of the impact of architectural changes — addition, removal, and replacement of components and connectors — on system dependability properties.

2. Tuning a dynamic system architecture based on dependability gauge readings will improve system dependability.

We have assumed that the most desirable balance of dependability and functionality in a system changes over time. In particular, when there is reason to believe that the system may be under attack, sacrificing some functionality for greater dependability may be desirable. We enhanced the SEAS prototype by making its architecture dynamic, so that it can be adapted to current conditions.

The evaluation experiments were conducted in two stages. In the first, the emphasis was on design-time analysis of the proposed static architecture for the next-generation SEAS. The objective was to determine whether, in this case, formal analysis of the architecture's dependability properties reveals any problems that the designers missed. The experimental measure of success for this first stage is the percentage of changes in which the designer-predicted and measured effects of the change on dependability properties are the same. In the second stage, two versions of the architecture would have been executed in parallel, one static and the other dynamic, with the latter instrumented using dependability gauges. The same attack is then run against both versions. The objective was to determine whether the instrumented version can continue to function longer — thanks to adapting intrusion resistance and intrusion tolerance mechanisms in response to the attack — than the static version. The experimental measure of success for the second stage is the ratio of the time (or a standardized measure of the number of attack steps) required to compromise the instrumented version to the time

[2]See the Web site `http://spinroot.com/spin/whatispin.html` for more information about Spin.

[3]See the Web page `http://www.ai.sri.com/~stickel/pttp.html` for more information about PTTP.

required to compromise the static version. The second planned experiment could not be performed, because of budget cuts.

Chapter 2

Project Overview

2.1 Motivation

It will soon be the case that most systems will be constructed, at least in part, from pre-existing components. The infrastructure needed to support a component-based lifecycle is currently emerging: intercomponent communication mechanisms (CORBA, DCOM) and data interchange formats (XML, DOM), service discovery mechanisms (Jini, e-Speak), and even higher-level collaboration and delegation mechanisms (SRI's Open Agent Architecture).

But a component-based lifecycle also poses new software engineering challenges. Most components developed for the commercial market will not be developed with the high dependability requirements of, e.g., DoD mission-critical applications in mind. So, if developers of highly dependable systems are to take maximal advantage of the availability of components, one question that must be answered is

> *How can a highly dependable system be built from components that may not be dependable?*

Basing systems on components will also increase the pace of system evolution. Components will quickly be declared obsolete, and replaced by new versions. As new versions of components, offering new capabilities, become available, users will naturally want to exploit those capabilities. Other pressures driving evolution — for example, the need to respond to changes in missions — can only intensify as well. Thus, another question that must be answered is

> *How can dependability be maintained when a system is constantly evolving?*

Previous SRI research on the design and construction of architectures for secure distributed transaction processing has shown how it is possible to build a secure system from not-necessarily-secure components. The primary innovation in our approach is to link an abstract architectural model that is proven secure to the implemented system architecture by a series of transformations that demonstrably preserve security. This verified link allows us to conclude that results obtained from our security analysis of the abstract model are applicable to the implementation as well. The same technique can be used to establish other system dependability properties.

For this effort, we have built upon this previous research,

- by generalizing our approach to dependability properties other than security and

- by using the transformation chains that link the abstract, analyzable architectural models — one per dependability property — to the concrete system architecture to dynamically update the abstract models as the running system evolves.

These additions made it possible to build *dependability gauges* that continuously measure dependability properties of an evolving system.

2.2 The Component-Based Lifecycle

In the future, systems will primarily be constructed by composing functionality provided by pre-existing components. These systems will be adapted to changes in requirements and available components by adding new functions, eliminating functions that are no longer needed, and substituting functions provided by new, improved components for those provided by obsolete components. The infrastructure required to support this component-based lifecycle is currently emerging. Intercomponent communication mechanisms, such as CORBA and DCOM, and data interchange formats, such as XML and DOM, are now widely used in both the defense and commercial sectors. Component discovery mechanisms, such as Sun's Jini and HP's e-Speak, while not yet in widespread use, are growing rapidly in popularity. And higher-level mechanisms for collaboration among components and delegation of service requests, such as SRI's Open Agent Architecture, are becoming available.

This new component-based lifecycle presents a host of new software engineering challenges. Traditionally, high-quality software is a product of a rigorously controlled, carefully monitored development process. Confidence that a system will satisfy its requirements is based on confidence that its subsystems satisfy their requirements. If those subsystems are, say, closed-source COTS components, what is the foundation for confidence in quality? Experience shows that developers of COTS components tend to value the addition of features more highly than quality, because most of their customers also value features more highly than quality. How, then, can such components be used in mission-critical systems that have strict dependability requirements?

The increased dynamism of the component-based lifecycle is another source of software engineering challenges. The life span of systems is measured in decades, while the life span of typical components can be as little as a few weeks. Frequent new releases of COTS components in particular must be expected: this is the principal source of revenue for the developers. Thus, even if system requirements never change, the system itself must be expected to undergo constant evolutionary change. And, of course, requirements do change, as a result of changes in missions, additional service demands from the users, needs for higher dependability, and so on. Even limited, infrequent, carefully controlled maintenance of systems tends to result in decreased quality due to unanticipated effects of changes to code. Frequent replacement of components by others that offer a somewhat different interface to somewhat different functionality — and that contain somewhat different bugs — will greatly exacerbate the traditional problems of maintenance, partly because of the increased frequency of change and partly because some of the traditional tools for coping with change (such as extensive regression testing of components and source code flow analysis) will no longer be available.

2.3 Dependable System Architectures

SRI research on methods for developing system architectures with guaranteed dependability properties can be applied to the software engineering challenges posed by the component-based lifecycle in several complementary ways. The main focus of our research has been on developing techniques for establishing that architecture transformations preserve dependability properties. This technology can be applied to the design of dependable architectures, to evaluating the dependability of component-based systems, and to the dependable evolution of component-based systems.

2.4 Designing for Dependability

There are two approaches to guaranteeing that a system satisfies a dependability constraint. The difference between these approaches can be illustrated by a simple example. Consider a system composed from components with varying security levels and clearances. Security policies for such multilevel systems include constraints on communication. For example, a typical constraint is that "read-up" is not allowed. That is, a component must not read data whose classification is greater than the component's clearance. How can we guarantee that such constraints are satisfied if the system contains closed-source components whose security has not been verified? One approach is to monitor all communication among components at runtime, and check whether the security policy is satisfied in each case. In cases where the communication would violate the security policy, the communication is blocked. An alternative approach is to design a system architecture that restricts communication among components so as to reduce the need for runtime constraint checking. If the architecture is designed so that communication channels between components exist only when communication between those components is consistent with the system's security policy, then no runtime checking of the security constraints is needed.

Our research has explored the latter approach. One product of that research is a dynamic architecture for secure distributed transaction processing (SDTP) [3, 10]. SDTP was designed by writing a simple abstract description of the architecture, showing that the description guarantees the desired security properties, successively refining the abstract description until a directly implementable concrete description results, and showing that each refinement step preserves satisfaction of the security policy. Refinement steps were the result of applying reusable refinement transformations that codify implementation techniques. Thus, SDTP is an example of how a dependable (in this case, secure) system can be constructed from not-necessarily-dependable components, without the overhead of runtime constraint checking.

Our current tools for supporting design of dependable architectures generalize capabilities in our earlier PegaSys system, which is currently in industrial use by software engineers with no particular training or experience in the use of formal methods. PegaSys appears to be an ordinary CASE tool, but it has an additional capability: it informs the user if any design constraints are violated. The application of formal methods to discover constraint violations is entirely "behind the scenes." When the current design toolset is fully mature, it too will be usable by practicing system designers.

2.5 Generating Dependability Gauge Readings

Formal methods, such as model checking and theorem proving, can be effective for determining whether systems have desired dependability properties. Less formal methods, such as simulation, also provide useful information about dependability, even when incapable of providing definitive answers. But, for complex systems, these methods cannot be applied directly. An abstract system model, designed specifically for the purposes of the particular dependability analysis, must be created. As much system detail as possible is abstracted away, in order to make the analysis more tractable.

Model checking failure tolerance of a system — where failure may be due to a transient accidental fault, a persistent accidental fault, or some form of malicious interference or corruption — provides a good illustration of the necessity of abstraction. Failure tolerance means that no combination of system state transitions and failure transitions can lead to a state where a system cannot supply essential services. Naïve models of complex systems will have too many states (often, infinitely many) for exhaustive state exploration to be feasible. This problem is particularly acute when closed-source components are involved, since such components' states potentially depend upon the entire history of their interactions with the system. Of course, for a failure tolerance analysis, a very simple model of such components is sufficient. Only the component's external interface protocol and whether it has failed or not is relevant. But, when a system is constructed from a large number of components with complex interface protocols in accordance with a complex architectural description, the number of states may still be too large for model checking. Boolean abstraction of protocols proves very useful in such cases, and SRI has developed techniques for automatically determining relevant predicates for use in the abstraction [4, 18].

In prior work, our application of abstraction techniques has been at design time. In the component-based lifecycle, where the system is constantly evolving, there are obvious advantages to applying them at runtime as well. For each dependability property of interest, an abstract system model that can be analyzed to determine whether the system has the property will be generated. The abstraction transformation steps will demonstrably co-preserve dependability. In other words, if the generated abstract model is dependable, then the more concrete input to the abstraction step must be dependable as well. As the system evolves, the abstract models will be updated by "replaying" the derivations that generated them. Thus, the dependability of the system can be dynamically reassessed whenever there are relevant changes in the environment or within the system itself. The results

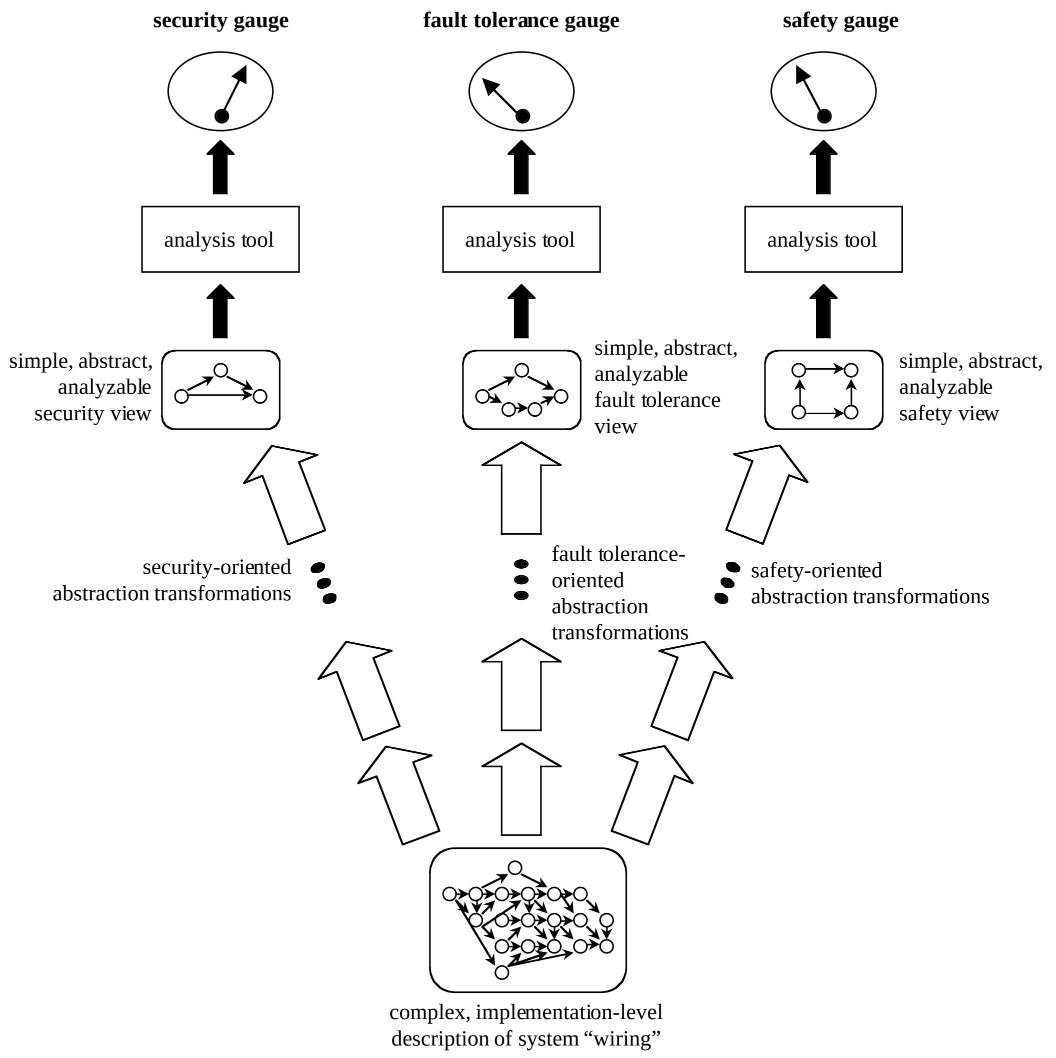
security gauge
fault tolerance gauge
safety gauge
analysis tool
analysis tool
analysis tool
simple, abstract, analyzable security view
simple, abstract, analyzable fault tolerance view
simple, abstract, analyzable safety view
security-oriented abstraction transformations
fault tolerance-oriented abstraction transformations
safety-oriented abstraction transformations
complex, implementation-level description of system "wiring"

Figure 2.1: Dependability Gauges

of analysis of the abstractions are displayed as dependability gauge readings, as shown in Figure 2.1.

2.6 Dependable Evolution through Perpetual Design

Much as transformations can be used at design time to create a baseline architecture that has desired dependability properties, transformations can be used at runtime to ensure continued dependability as requirements, the system, and its environment evolve. The basic idea is that changes in dependability gauge readings can trigger application of dependability-enhancing, functionality-preserving architecture transformations.

Any of the following may trigger re-architecting.

- *A change in required dependability level.* The system administrator may decide that the system should be made more (or less) secure, more (or less) fault tolerant, more (or less) safe, etc. Architectural transformations expected to have the appropriate effect will be applied, and the results monitored using the appropriate dependability gauge, as shown in Figure 2.2. Other dependability gauges will be monitored as well, to ensure that other dependability constraints remain satisfied.

- *A change in desired functionality.* The system user may request that the system provide some additional service, requiring the integration of additional components or the replacement of components with other components offering additional functionality. If a new component is added to the system, or one component is replaced by another that proves less dependable, then the system architecture may have to be made more dependable to compensate. An example is provided by our SDTP work, where we showed that a very simple architecture can be used to securely integrate single-level databases that are all at the same level, but substantial architectural complexity must be introduced when a database at a different level is integrated into the system.

- *A change in component availability.* Changes in component availability can also necessitate addition and replacement of system components, and hence changes in system architecture.

- *A change in the system's behavior or performance.* Monitoring of dependability gauges may reveal dependability requirements that were being satisfied but are no longer being satisfied. Similarly, monitoring of performance gauges may reveal that system performance has declined to an unsatisfactory level. Architectural transformations can be applied to address such problems.

So, there are at least two advantages to designing an architecture that ensures satisfaction of dependability requirements without runtime constraint checking.

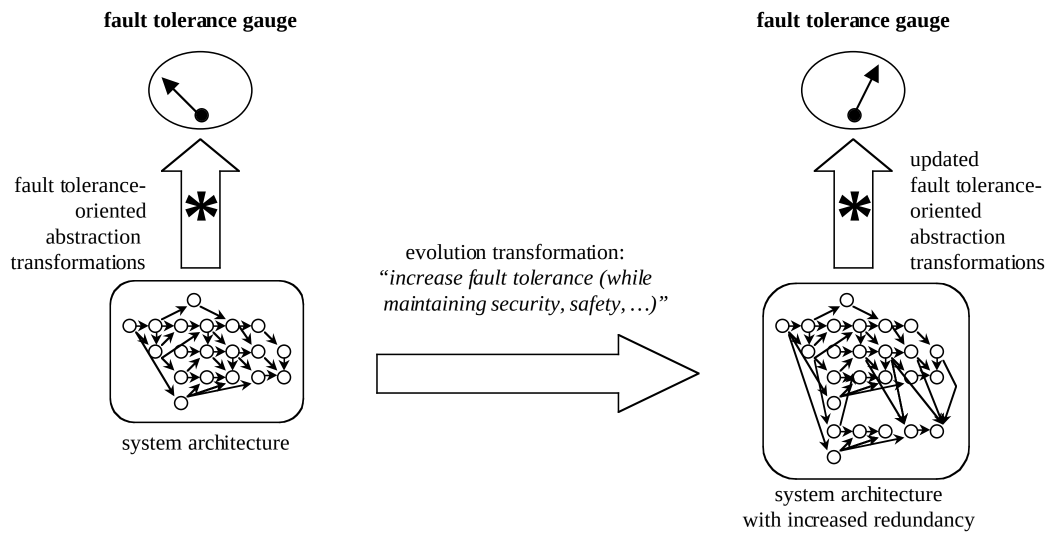
fault tolerance gauge
fault tolerance-oriented abstraction transformations
system architecture
evolution transformation: "increase fault tolerance (while maintaining security, safety, …)"
fault tolerance gauge
updated fault tolerance-oriented abstraction transformations
system architecture with increased redundancy

Figure 2.2: Re-Architecting to Improve Fault Tolerance

First, the runtime overhead of checking is eliminated. Second, runtime checking does not seem well suited to guaranteeing that some types of dependability constraints are satisfied. The primary problem with "designed-in" dependability is coping with change. A system may be dependable when first fielded, but how can dependability be assured as it evolves? Our answer to this question is: Use the same technology that was used to assure dependability of the initial release. This means maintaining the links between the abstract models and the implementation as the system evolves, and re-analyzing the abstract models whenever they are altered. In other words, we want to bring the design-time capabilities we have developed for assuring dependability to runtime, resulting in a system lifecycle based on *perpetual design for perpetual dependability*.

In the following chapters, the gauge-building technology and the two more complex of the four sample gauges developed for this effort will be discussed in greater detail.

Chapter 3

Gauge-Building Technology

3.1 Abstraction Transformation Rule Grammar

Teal is a pattern language for architectural descriptions. For example, the Teal specifcation

```
acme_channel_to_sadl_var: PATTERN
  BEGIN
    ABSTRACT_TEMPLATE(Acme)::
      system @m = {}
        component @f1 = { ports {@op; @@ops1; @@ips1} };
        component @f2 = { ports {@ip; @@ops2; @@ips2} };
        connector @c = { roles {@ir; @or}};
        @@restc;
        attachments {
          @c.@ir to @f1.@op;
          @c.@or to @f2.@ip;
          @@resta}
      }
    END_TEMPLATE
    CONCRETE_TEMPLATE(Sadl)::
      @m: ARCHITECTURE [ @@ips -> @@ops ]
      BEGIN
        @f1: Functional_style!Function[ @@ips1 -> @@ops1 ]
        @f2: Functional_style!Function[ @@ips2 -> @@ops2 ]
        @v:  Shared_Memory_style!Variable(@t)
        @@restc
        @a1: CONSTRAINT = Shared_Memory_style!Writes(@f1, @v)
        @a2: CONSTRAINT = Shared_Memory_style!Reads(@f2, @v)
        @@resta
      END
    END_TEMPLATE
    ASSOCIATIONS::
      @op --> ()
      @ip --> ()
      @ir --> ()
      @or --> ()
      @c  --> (@v)
  END
```

says that any match of an Acme architectural description and SADL architectural description [11] to the abstract and concrete templates, respectively, is an instance of the pattern.[1] (This pattern says, roughly, that an Acme connector that connects two Acme components is an abstraction of a SADL shared variable that is written to by one SADL component and read by another, and, conversely, that the SADL shared variable is a way of implementing the Acme connector.)

A Teal pattern can be compiled into a pair of refinement rules: a rule for generating a match to the concrete pattern given a match to the abstract pattern (i.e., a refinement rule), and a rule for generating a match to the abstract pattern given a match to the concrete pattern (i.e., an abstraction rule).

Since, for the present effort, our only concern is with abstraction of Acme descriptions — Acme is the *lingua franca* of DASADA — the "refinement" direction and the indication of the architectural description language used is irrelevant, and the following simpler notation, expressed in Prolog grammar rules, for abstraction transformations has been employed.[2]

```
xform(xform(Id, InDesc, Map, true, OutDesc)) -->
        [token(transformation, key)],
        id_or_meta(Id),
        [token(from, key)],
        architectural_description(InDesc),
        [token(to, key)],
        architectural_description(OutDesc),
        [token(where, key)],
        map(Map).

map(map(Id, Maplets)) -->
```

[1]The details of pattern matching will be left vague, deliberately, as the process is somewhat complicated. Ideally, from a purely technical point of view, the matching should be performed on the abstract syntax trees. But, in that case, the user must understand the details of the internal representation of the abstract syntax. The REFINE language [6, p. 313ff] provides some nice illustrations of the complexities that result from this level of precision. On the other hand, a straightforward match to the surface syntax of the descriptions is entirely unsatisfactory, since it makes irrelevant features of the description, such as the order of declarations, relevant. Teal attempts to develop a "user friendly" compromise between the two, which seems to satisfy users' intuitions quite well, in the sense that they find it easy to write Teal patterns that correctly express their intentions without any understanding of the formal grammar of architecture description languages. However, the details of the algorithm are best understood by examining the Teal code for pattern matching.

[2]The nonterminals `id_or_meta`, `architectural_description`, `startblock`, `endblock`, `opt_semi`, `segment_metavar`, and `attribute_value` are defined in the Acme grammar, below.

```
        [token(map, key)],
        id_or_meta(Id),
        [token(with, key)],
        startblock,
        maplet_list(Maplets),
        endblock.

maplet_list([Maplet|Maplets]) -->
        maplet(Maplet),
        [token(';', misc)],
        maplet_list(Maplets).
maplet_list([Maplet]) -->
        maplet(Maplet),
        opt_semi.

maplet(maplet([From], [To])) -->
        segment_metavar(From),
        [token(to, key)],
        segment_metavar(To).
maplet(maplet(From, To)) -->
        attribute_value(From),
        [token(to, key)],
        attribute_value(To).
```

An additional benefit of the exclusive concentration on abstraction is that the pattern variable notation can be extended to give meaningful names to “new” objects introduced at the abstract level. For example, the `@.(@c,@n)` indicates that the name for the new object bound to this variable should be the result of joining the name for the object bound `@c` and the name for the object bound to `@n` by a dot, resulting in an Acme “qualified name”. Typically, this sort of variable is used when `@c` is bound to a component, and `@n` is bound to some object that is uniquely identified by its name within the component, but not globally. Similarly, `@&(@n1,@2)` indicates that names should be joined by `_and_`, and `@_(@n1,@n2)` that they should be joined by an underscore. (Also, `@*(@@list1,@@list2)` is a variable bound to the result of appending the list of objects that is bound to `@@list1` to the list of objects that is bound to `@@list2`.)

As the above example of a Teal pattern illustrates, the bulk of the abstraction pattern grammar required is a grammar for Acme, extended with pattern variables. The following is our extended Acme grammar, expressed in terms of Prolog grammar rules, that we employed. It is based on a grammar written by Fred Gilham of SRI (`gilham@csl.sri.com`) for use with a recursive descent parser implmented in Common Lisp that is part of the design-time portion of the Teal toolkit. Gilham’s grammar was based in turn upon a Popart grammar for Acme (without metavariables) written by Dave Wile (`wile@teknowledge.com`), one of the principal designers of Acme.

```
architectural_description(architectural_description(MdeclL, SysL)) -->
        meta_declaration_list(MdeclL),
        system_list(SysL).

meta_declaration_list([MDecl]) -->
        meta_declaration(MDecl).
meta_declaration_list([MDecl|MdeclL]) -->
        meta_declaration(MDecl),
        [token(’;’, misc)],
        meta_declaration_list(MdeclL).
meta_declaration_list([]) --> [].

meta_declaration(Id) -->
        segment_metavar(Id).
meta_declaration(TypDecl) -->
        type_declaration(TypDecl).
meta_declaration(Templt) -->
        template(Templt).
meta_declaration(Fam) -->
        family(Fam).
meta_declaration(Prop) -->
        property(Prop).
meta_declaration(Props) -->
        properties(Props).


family(family(TypedId, Formals, DesignRuleEltL)) -->
      [token(family, key)],
      typed_identifier(TypedId),
      formal_parameters_or_epsilon(Formals),
      [token(’=’, misc)],
      startblock,
      design_rules_element_list(DesignRuleEltL),
      endblock.

typed_identifier(Id) -->
      segment_metavar(Id).
typed_identifier(typed_identifier(IdM, TypNam)) -->
      id_or_meta(IdM),
      [token(’:’, misc)],
      type_name(TypNam).
typed_identifier(untyped_identifier(IdM)) -->
      id_or_meta(IdM).

typed_identifier_or_epsilon(TypedId) -->
      typed_identifier(TypedId).
typed_identifier_or_epsilon --> [].

typed_identifier_list([TypedId|TypedIdL]) -->
      typed_identifier(TypedId),
      [token(’,’, misc)],
      typed_identifier_list(TypedIdL).
typed_identifier_list([TypedId]) -->
      typed_identifier(TypedId).

type_name_list([TypNam]) -->
      type_name(TypNam).
```

```
type_name_list([TypNam|TypNamL]) -->
      type_name(TypNam),
      [token(',', misc)],
      type_name_list(TypNamL).

type_name(Id) -->
      segment_metavar(Id).
type_name(Nam) -->
      acme_name(Nam).
type_name(CatLit) -->
      category_literal(CatLit).
type_name(PrimPropTyp) -->
      primitive_property_type(PrimPropTyp).
type_name(IdM) -->
      id_or_meta(IdM).

formal_parameters_or_epsilon(FormalL) -->
      [token('(', misc)],
      formal_parameter_list(FormalL),
      [token(')', misc)].
formal_parameters_or_epsilon([]) -->
      [token('(', misc)],
      [token(')', misc)].
formal_parameters_or_epsilon([]) --> [].

formal_parameter_list([Formal|FormalL]) -->
      formal_parameter(Formal),
      [token(',', misc)],
      formal_parameter_list(FormalL).
formal_parameter_list([Formal]) -->
      formal_parameter(Formal).


formal_parameter(Id) -->
      segment_metavar(Id).
formal_parameter(formal_parameter(IdML, SynClass)) -->
      id_or_meta_list(IdML),
      [token(':', misc)],
      syntactic_class(SynClass).

design_rules_element_list([DesignRulElt|DesignRulEltL]) -->
      design_rules_element(DesignRulElt),
      [token(';', misc)],
      design_rules_element_list(DesignRulEltL).
design_rules_element_list([DesignRulElt]) -->
      design_rules_element(DesignRulElt),
      opt_semi.

design_rules_element(Id) -->
      segment_metavar(Id).
design_rules_element(TypDecl) -->
      type_declaration(TypDecl).
design_rules_element(Templt) -->
      template(Templt).
design_rules_element(Constrs) -->
      constraints(Constrs).
design_rules_element(Harn) -->
```

```
      harness(Harn).

template(UntypdTemplt) -->
      untyped_template(UntypdTemplt).
template(ComptTemplt) -->
      component_template(ComptTemplt).
template(ConnctrTemplt) -->
      connector_template(ConnctrTemplt).
template(PortTemplt) -->
      port_template(PortTemplt).
template(RoleTemplt) -->
      role_template(RoleTemplt).

untyped_template(untyped_template(TypedId, FormalL, CatLits, Defn)) -->
      [token(template, key)],
      typed_identifier(TypedId),
      formal_parameters_or_epsilon(FormalL),
      category_literals(CatLits),
      [token('=', misc)],
      definition(Defn).

category_literals(category_literals(CatLitPlus)) -->
      [token(defining, key)],
      [token('(', misc)],
      category_literal_list(CatLitPlus),
      [token(')', misc)].
category_literals(category_literals([])) --> [].
category_literals(category_literals_token(components)) --> [token(components, key)].
category_literals(category_literals_token(connectors)) --> [token(connectors, key)].
category_literals(category_literals_token(ports))      --> [token(ports, key)].
category_literals(category_literals_token(roles))      --> [token(roles, key)].
category_literals(category_literals_token(properties)) --> [token(properties, key)].

category_literal_list([CatLitNam]) -->
      category_literal_name(CatLitNam).
category_literal_list([CatLitNam|CatLitPlus]) -->
      category_literal_name(CatLitNam),
      [token(',', misc)],
      category_literal_list(CatLitPlus).


definition(definition(AggDesc)) -->
      aggregate_description(AggDesc).

aggregate_description(aggregate_description(DeclOrHarnL)) -->
      startblock,
      declaration_or_harness_list(DeclOrHarnL),
      endblock.

declaration_or_harness_list([DeclOrHarn]) -->
      declaration_or_harness(DeclOrHarn),
      opt_semi.
declaration_or_harness_list([DeclOrHarn|DeclOrHarnL]) -->
      declaration_or_harness(DeclOrHarn),
      [token(';', misc)],
      declaration_or_harness_list(DeclOrHarnL).
```

```
declaration_or_harness(Id) -->
      segment_metavar(Id).
declaration_or_harness(Decl) -->
      declaration(Decl).
declaration_or_harness(Harn) -->
      harness(Harn).

harness(harness(AggDesc)) -->
      [token(harness, key)],
      aggregate_description(AggDesc).

syntactic_class(syntactic_class(CatLitt)) -->
      category_literal_type(CatLitt).
syntactic_class(syntactic_class(CatLit)) -->
      category_literal(CatLit).
syntactic_class(syntactic_class(CatLits)) -->
      category_literals(CatLits).
syntactic_class(syntactic_class(TypNam)) -->
      type_name(TypNam).

category_literal_type(category_literal_type(CatLit, TypNam)) -->
      category_literal(CatLit),
      type_name(TypNam).

category_literal_name(Id) -->
      segment_metavar(Id).
category_literal_name(category_literal_name(CatLit, Nam)) -->
      category_literal(CatLit),
      acme_name(Nam).

category_literal(category_literal(component)) --> [token(component, key)].
category_literal(category_literal(connector)) --> [token(connector, key)].
category_literal(category_literal(port))      --> [token(port, key)].
category_literal(category_literal(role))      --> [token(role, key)].
category_literal(category_literal(property))  --> [token(property, key)].

multiple(multiple_with_template(ConstL, TempltInv)) -->
      [token(multiple, key)],
      constituent_list(ConstL),
      [token('=', misc)],
      template_invocation(TempltInv).
multiple(multiple_with_name(ConstL, Nam)) -->
      [token(multiple, key)],
      [token('(', misc)],
      constituent_list(ConstL),
      [token(')', misc)],
      [token('=', misc)],
      acme_name(Nam).

constituents([Const]) -->
      constituent(Const).
constituents([Const|Consts]) -->
      constituent(Const),
      [token(',', misc)],
      constituents(Consts).
```

```
template_invocation(template_invocation(Nam, ParamL)) -->
      acme_name(Nam),
      [token('(', misc)],
      actual_parameter_list(ParamL),
      [token(')', misc)].

actual_parameters([Param]) -->
      actual_parameter(Param).
actual_parameters([Param|Params]) -->
      actual_parameter(Param),
      [token(',', misc)],
      actual_parameters(Params).

actual_parameter(Id) -->
      segment_metavar(Id).
actual_parameter(actual_parameter(Con)) -->
      constituent_or_name(Con).
actual_parameter(actual_parameter(ActL)) -->
      actual_list(ActL).
actual_parameter(actual_parameter(AggDesc)) -->
      aggregate_description(AggDesc).

constituent_or_name(Id) -->
      segment_metavar(Id).
constituent_or_name(ConOrNam) -->
      constituent(ConOrNam).
constituent_or_name(TempltInv) -->
      template_invocation(TempltInv).
constituent_or_name(Nam) -->
      acme_name(Nam).

constituent(Id) -->
      segment_metavar(Id).
constituent(constituent(Compt)) -->
      component(Compt).
constituent(constituent(Conctr)) -->
      connector(Conctr).
constituent(constituent(Port)) -->
      port(Port).
constituent(constituent(Role)) -->
      role(Role).

actual_list(actual_list(ConstrL)) -->
      constraint_list(ConstrL).
actual_list(actual_list(ConstL)) -->
      constituent_list(ConstL).

constraint_list(MixL) -->
      mixed_list(MixL).
constraint_list(Rep) -->
      representation_s(Rep).
constraint_list(Prop) -->
      property_s(Prop).

mixed_list(mixed_list(Constrs)) -->
      startblock,
      single_constraints(Constrs),
      endblock.

single_constraints(Constr) -->
      single_constraint(Constr).
single_constraints([Constr|Constrs]) -->
      single_constraint(Constr),
      [token(';', misc)],
      %% !,
      single_constraints(Constrs).

single_constraint(Id) -->
      segment_metavar(Id).
single_constraint(constraint(Rep)) -->
      representation(Rep).
single_constraint(constraint(Prop)) -->
      property(Prop).

constituent_list(ConstOrNams) -->
      startblock,
      constituent_or_names(ConstOrNams),
      endblock.

constituent_or_names(ConstOrNam) -->
      constituent_or_name(ConstOrNam).
constituent_or_names([ConstOrNam|ConstOrNams]) -->
      constituent_or_name(ConstOrNam),
      [token(';', misc)],
      %% !,
      constituent_or_names(ConstOrNams).


description(ComptDesc) -->
      component_description(ComptDesc).
description(ConctrDesc) -->
      connector_description(ConctrDesc).

type_declaration(CmpTypDecl) -->
      component_type_declaration(CmpTypDecl).
type_declaration(ConctrTypDecl) -->
      connector_type_declaration(ConctrTypDecl).
type_declaration(PortTypDecl) -->
      port_type_declaration(PortTypDecl).
type_declaration(RTypDecl) -->
      role_type_declaration(RTypDecl).
type_declaration(PropTypDecl) -->
      property_type_declaration(PropTypDecl).

system_list([Sys]) -->
      system(Sys),
      opt_semi.
system_list([Sys|SysL]) -->
      system(Sys),
      [token(';', misc)],
      !,
      system_list(SysL).

system(system(TypedId, DeclL)) -->
```

```
      [token(system, key)],
      typed_identifier(TypedId),
      [token('=', misc)],
      startblock,
      declaration_list(DeclL),
      endblock.
system(system_without_declaration(TypedId, [])) -->
      [token(system, key)],
      typed_identifier(TypedId).

declaration_list([Decl|DeclL]) -->
      declaration(Decl),
      [token(';', misc)],
      declaration_list(DeclL).
declaration_list([Decl]) -->
      declaration(Decl),
      opt_semi.

declaration(Id) -->
      segment_metavar(Id).
declaration(Compt) -->
      component_s(Compt).
declaration(Conctr) -->
      connector_s(Conctr).
declaration(Role) -->
      role_s(Role).
declaration(Port) -->
      port_s(Port).
declaration(Mult) -->
      multiple(Mult).
declaration(AttDecl) -->
      attachments_declaration(AttDecl).
declaration(Constrs) -->
      constraints(Constrs).
declaration(TempltInv) -->
      template_invocation(TempltInv).

component_template(component_template_defining_constituent(IdM, OptFormal, Const, Defn)) -
->
      [token(component, key)],
      [token(template, key)],
      id_or_meta(IdM),
      formal_parameters_or_epsilon(OptFormal),
      [token(defining, key)],
      !,
      [token('(', misc)],
      constituent(Const),
      [token(')', misc)],
      [token('=', misc)],
      definition(Defn).
component_template(component_template(IdM, OptFormal, Defn)) -->
      [token(component, key)],
      [token(template, key)],
      id_or_meta(IdM),
      formal_parameters_or_epsilon(OptFormal),
      [token('=', misc)],
      !,
      definition(Defn).

component_type_declaration(component_type_extension_declaration(TypNam, Nam, Compt-
Desc)) -->
      [token(component, key)],
      [token(type, key)],
      acme_name(TypNam),
      [token(extends, key)],
      !,
      acme_name(Nam),
      [token(with, key)],
      component_description(ComptDesc).
component_type_declaration(component_type_declaration(TypNam, ComptDesc)) -->
      [token(component, key)],
      [token(type, key)],
      acme_name(TypNam),
      [token('=', misc)],
      !,
      component_description(ComptDesc).
component_type_declaration(unidentified_component_type_declaration(TypNam)) -->
      [token(component, key)],
      [token(type, key)],
      acme_name(TypNam).

component_s(Cmp) -->
      component(Cmp).
component_s(Compt) -->
      components(Compt).


component(component_with_description(TypedId, ComptDesc)) -->
      [token(component, key)],
      typed_identifier(TypedId),
      [token('=', misc)],
      !,
      component_description(ComptDesc).
component(unidentified_component(TypedId)) -->
      [token(component, key)],
      typed_identifier(TypedId).

components(components(ComptL)) -->
      [token(components, key)],
      startblock,
      multiple_components_list(ComptL),
      endblock.

multiple_components_list([Compt|ComptL]) -->
      multiple_components(Compt),
      [token(';', misc)],
      multiple_components_list(ComptL).
multiple_components_list([Compt]) -->
      multiple_components(Compt),
      opt_semi.

multiple_components(Id) -->
      segment_metavar(Id).
multiple_components(component_list_with_decl(TypedIdL, ComptDesc)) -->
```

```
      typed_identifier_list(TypedIdL),
      [token('=', misc)],
      component_description(ComptDesc).
multiple_components(unidentified_component_list(TypedIdL)) -->
      typed_identifier_list(TypedIdL).

component_description(TempltInv) -->
      template_invocation(TempltInv).
component_description(ComptDefn) -->
      component_definition(ComptDefn).
component_description(ComptInst) -->
      component_instantiation(ComptInst).


component_instantiation(component_instantiation_with_extension(TypNamL, ComptDesc)) -
->
      [token(new, key)],
      type_name_list(TypNamL),
      [token(extended, key)],
      [token(with, key)],
      component_description(ComptDesc).
component_instantiation(component_instantiation(TypNamL)) -->
      [token(new, key)],
      type_name_list(TypNamL).

component_definition(component_definition(PortOrConstrsL)) -->
      startblock,
      port_s_or_constraints_list(PortOrConstrsL),
      endblock.

port_s_or_constraints_list([PortOrConstrs|PortOrConstrsL]) -->
      port_s_or_constraints(PortOrConstrs),
      [token(';', misc)],
      port_s_or_constraints_list(PortOrConstrsL).
port_s_or_constraints_list([PortOrConstrs]) -->
      port_s_or_constraints(PortOrConstrs),
      opt_semi.

port_s_or_constraints(Id) -->
      segment_metavar(Id).
port_s_or_constraints(Port) -->
      port_s(Port).
port_s_or_constraints(Constrs) -->
      constraints(Constrs).

connector_template(connector_template_with_constituent(IdM, OptFormal, Const, Defn)) -
->
      [token(connector, key)],
      [token(template, key)],
      id_or_meta(IdM),
      formal_parameters_or_epsilon(OptFormal),
      [token(defining, key)],
      !,
      [token('(', misc)],
      constituent(Const),
      [token(')', misc)],
      [token('=', misc)],
      definition(Defn).
connector_template(connector_template(IdM, OptFormal, Defn)) -->
      [token(connector, key)],
      [token(template, key)],
      id_or_meta(IdM),
      formal_parameters_or_epsilon(OptFormal),
      [token('=', misc)],
      definition(Defn).

connector_type_declaration(connector_type_declaration_with_extension(TypNam, Nam, Con-
nDesc)) -->
      [token(connector, key)],
      [token(type, key)],
      acme_name(TypNam),
      [token(extends, key)],
      !,
      acme_name(Nam),
      [token(with, key)],
      connector_description(ConnDesc).
connector_type_declaration(connector_type_declaration(TypNam, ConnDesc)) -->
      [token(connector, key)],
      [token(type, key)],
      acme_name(TypNam),
      [token('=', misc)],
      !,
      connector_description(ConnDesc).
connector_type_declaration(unidentified_connector_type_declaration(TypNam)) -->
      [token(connector, key)],
      [token(type, key)],
      acme_name(TypNam).

connector_s(Conn) -->
      connector(Conn).
connector_s(ConnL) -->
      connectors(ConnL).

connector(connector(TypedId, ConnDesc)) -->
      [token(connector, key)],
      typed_identifier(TypedId),
      [token('=', misc)],
      !,
      connector_description(ConnDesc).
connector(unidentified_connector(TypedId)) -->
      [token(connector, key)],
      typed_identifier(TypedId).

connectors(connectors(ConnsL)) -->
      [token(connectors, key)],
      startblock,
      multiple_connectors_list(ConnsL),
      endblock.

multiple_connectors_list([Conns|ConnsL]) -->
      multiple_connectors(Conns),
      [token(';', misc)],
      multiple_connectors_list(ConnsL).
multiple_connectors_list([Conns]) -->
```

```
      multiple_connectors(Conns),
      opt_semi.

multiple_connectors(Id) -->
      segment_metavar(Id).
multiple_connectors(multiple_connectors_with_description(TypedIdL, ConnDesc)) -->
      typed_identifier_list(TypedIdL),
      [token('=', misc)],
      connector_description(ConnDesc).
multiple_connectors(multiple_connectors(TypedIdL)) -->
      typed_identifier_list(TypedIdL).

connector_description(TempltInv) -->
      template_invocation(TempltInv).
connector_description(ConnDefn) -->
      connector_definition(ConnDefn).
connector_description(ConnInst) -->
      connector_instantiation(ConnInst).


connector_instantiation(connector_instantiation_with_description(TypNamL, ConnDesc)) -
->
      [token(new, key)],
      type_name_list(TypNamL),
      [token(extended, key)],
      [token(with, key)],
      connector_description(ConnDesc).
connector_instantiation(connector_instantiation(TypNamL)) -->
      [token(new, key)],
      type_name_list(TypNamL).

connector_definition(connector_definition(RoleOrConnL)) -->
      startblock,
      role_s_or_constraints_list(RoleOrConnL),
      endblock.

role_s_or_constraints_list([RoleOrConsts|RoleOrConstsL]) -->
      role_s_or_constraints(RoleOrConsts),
      role_s_or_constraints_list(RoleOrConstsL).
role_s_or_constraints_list([RoleOrConsts]) -->
      role_s_or_constraints(RoleOrConsts).

role_s_or_constraints(Id) -->
      segment_metavar(Id).
role_s_or_constraints(Role) -->
      role_s(Role).
role_s_or_constraints(Consts) -->
      constraints(Consts).

port_template(port_template_with_definition(IdM, OptFormal, Const, Defn)) -->
      [token(port, key)],
      [token(template, key)],
      id_or_meta(IdM),
      formal_parameters_or_epsilon(OptFormal),
      [token(defining, key)],
      !,
      [token('(', misc)],
      constituent(Const),
      [token(’)’, misc)],
      [token(’=’, misc)],
      definition(Defn).
port_template(port_template(IdM, OptFormal, Defn)) -->
      [token(port, key)],
      [token(template, key)],
      id_or_meta(IdM),
      formal_parameters_or_epsilon(OptFormal),
      [token(’=’, misc)],
      definition(Defn).

port_type_declaration(port_type_declaration_with_extension(TypNam, Nam, PortDesc)) -
->
      [token(port, key)],
      [token(type, key)],
      acme_name(TypNam),
      [token(extends, key)],
      !,
      acme_name(Nam),
      [token(with, key)],
      port_description(PortDesc).
port_type_declaration(port_type_declaration(TypNam, PortDesc)) -->
      [token(port, key)],
      [token(type, key)],
      acme_name(TypNam),
      [token(’=’, misc)],
      !,
      port_description(PortDesc).
port_type_declaration(unidentified_port_type_declaration(TypNam)) -->
      [token(port, key)],
      [token(type, key)],
      acme_name(TypNam).

port_s(Port) -->
      port(Port).
port_s(Ports) -->
      ports(Ports).


port(port(TypedId, PortDesc)) -->
      [token(port, key)],
      typed_identifier(TypedId),
      [token(’=’, misc)],
      !,
      port_description(PortDesc).
port(unidentified_port(TypedId)) -->
      [token(port, key)],
      typed_identifier(TypedId).

ports(ports(PortsL)) -->
      [token(ports, key)],
      startblock,
      multiple_ports_list(PortsL),
      endblock.

multiple_ports_list([Ports|PortsL]) -->
```

```
      multiple_ports(Ports),
      [token(';', misc)],
      multiple_ports_list(PortsL).
multiple_ports_list([Ports]) -->
      multiple_ports(Ports),
      opt_semi.

multiple_ports(Id) -->
      segment_metavar(Id).
multiple_ports(multiple_ports(TypedIdL, PortDesc)) -->
      typed_identifier_list(TypedIdL),
      [token('=', misc)],
      port_description(PortDesc).
multiple_ports(unidentified_multiple_ports(TypedIdL)) -->
      typed_identifier_list(TypedIdL).

port_description(TempltInv) -->
      template_invocation(TempltInv).
port_description(PortDefn) -->
      port_definition(PortDefn).
port_description(PortInst) -->
      port_instantiation(PortInst).

port_instantiation(port_instantiation(TypNamL, PortDesc)) -->
      [token(new, key)],
      type_name_list(TypNamL),
      [token(extended, key)],
      [token(with, key)],
      port_description(PortDesc).
port_instantiation(port_instantiation(TypNamL)) -->
      [token(new, key)],
      type_name_list(TypNamL).

port_definition(port_definition(ConstrL)) -->
      startblock,
      constraints_list(ConstrL),
      endblock.

constraints_list([Cn]) -->
      constraints(Cn),
      opt_semi.
constraints_list([Cn|Cns]) -->
      constraints(Cn),
      [token(';', misc)],
      constraints_list(Cns).

role_template(role_template_with_definition(IdM, OptFormal, Const, Defn)) -->
      [token(role, key)],
      [token(template, key)],
      id_or_meta(IdM),
      formal_parameters_or_epsilon(OptFormal),
      [token(defining, key)],
      !,
      [token('(', misc)],
      constituent(Const),
      [token(')', misc)],
      [token('=', misc)],
      definition(Defn).
role_template(role_template(IdM, OptFormal, Defn)) -->
      [token(role, key)],
      [token(template, key)],
      id_or_meta(IdM),
      formal_parameters_or_epsilon(OptFormal),
      [token('=', misc)],
      definition(Defn).

role_type_declaration(role_type_declaration_with_extension(TypNam, Nam, Cd)) -->
      [token(role, key)],
      [token(type, key)],
      acme_name(TypNam),
      [extends],
      !,
      acme_name(Nam),
      [token(with, key)],
      role_description(Cd).
role_type_declaration(role_type_declaration(TypNam, RoleDesc)) -->
      [token(role, key)],
      [token(type, key)],
      acme_name(TypNam),
      [token('=', misc)],
      !,
      role_description(RoleDesc).
role_type_declaration(unidentified_role_type_declaration(TypNam)) -->
      [token(role, key)],
      [token(type, key)],
      acme_name(TypNam).

role_s(Role) -->
      role(Role).
role_s(Roles) -->
      roles(Roles).


role(role(TypedId, RoleDesc)) -->
      [token(role, key)],
      typed_identifier(TypedId),
      [token('=', misc)],
      !,
      role_description(RoleDesc).
role(unidentified_role(TypedId)) -->
      [token(role, key)],
      typed_identifier(TypedId).

roles(roles(RolesL)) -->
      [token(roles, key)],
      startblock,
      multiple_roles_list(RolesL),
      endblock.

multiple_roles_list([Roles|RolesL]) -->
      multiple_roles(Roles),
      [token(';', misc)],
      multiple_roles_list(RolesL).
multiple_roles_list([Roles]) -->
```

```
      multiple_roles(Roles),
      opt_semi.

multiple_roles(Id) -->
      segment_metavar(Id).
multiple_roles(multiple_roles(TypedIdL, RoleDesc)) -->
      typed_identifier_list(TypedIdL),
      [token('=', misc)],
      role_description(RoleDesc).
multiple_roles(unidentified_multiple_roles(TypedIdL)) -->
      typed_identifier_list(TypedIdL).

role_description(TempltInv) -->
      template_invocation(TempltInv).
role_description(RoleDefn) -->
      role_definition(RoleDefn).
role_description(RoleInst) -->
      role_instantiation(RoleInst).

role_instantiation(role_instantiation(TypNamL, RoleDesc)) -->
      [token(new, key)],
      type_name_list(TypNamL),
      [token(extended, key)],
      [token(with, key)],
      role_description(RoleDesc).
role_instantiation(role_instantiation(TypNamL)) -->
      [token(new, key)],
      type_name_list(TypNamL).

role_definition(role_definition(ConstL)) -->
      startblock,
      constraints_list(ConstL),
      endblock.

attachments_declaration(Attach) -->
      attachments_decl(Attach).
attachments_declaration(Attach) -->
      attachment_decl(Attach).

attachment_decl(attachment_decl(Attach)) -->
      [token(attachment, key)],
      attachment(Attach).

attachments_decl(attachments_decl(Attach)) -->
      [token(attachments, key)],
      startblock,
      attachment_or_constraint_list(Attach),
      endblock.

attachment_or_constraint_list([Attach]) -->
      attachment_or_constraints(Attach),
      opt_semi.
attachment_or_constraint_list([Attach|AttachL]) -->
      attachment_or_constraints(Attach),
      [token(';', misc)],
      attachment_or_constraint_list(AttachL).
```

```
attachment_or_constraints(Id) -->
      segment_metavar(Id).
attachment_or_constraints(Attach) -->
      attachment(Attach).
attachment_or_constraints(ConstrL) -->
      constraints(ConstrL).

attachment(attachment(PortNam, RoleNam)) -->
      acme_name(PortNam),
      [token(to, key)],
      acme_name(RoleNam).

constraints(Id) -->
      segment_metavar(Id).
constraints(Prop) -->
      property_s(Prop).
constraints(Rep) -->
      representation_s(Rep).

representation_s(Rep) -->
      representation(Rep).
representation_s(Reps) -->
      representations(Reps).

representation(representation(Id, SingRep)) -->
      [token(representation, key)],
      id_or_meta(Id),
      [token('=', misc)],
      !,
      single_representation(SingRep).
representation(unnamed_representation(SingRep)) -->
      [token(representation, key)],
      single_representation(SingRep).

representations(representations(RepL)) -->
      [token(representations, key)],
      startblock,
      representation_list(RepL),
      endblock.

representation_list([Rep|RepL]) -->
      single_representation(Rep),
      [token(';', misc)],
      representation_list(RepL).
representation_list([Rep]) -->
      single_representation(Rep),
      opt_semi.


single_representation(Id) -->
      segment_metavar(Id).
single_representation(single_representation(Sys, BindMapDefn)) -->
      startblock,
      system(Sys),
      bindings_map_definition(BindMapDefn),
      endblock.
single_representation(unbound_single_representation(Sys)) -->
```

```
      startblock,
      system(Sys),
      endblock.

abstraction_map(BindMapDefn) -->
      bindings_map_definition(BindMapDefn).
abstraction_map(Omd) -->
      other_map_definition(Omd).
abstraction_map([]) --> [].


property_s_list([Prop|PropL]) -->
      property_s(Prop),
      [token(';', misc)],
      property_s_list(PropL).
property_s_list([Prop]) -->
      property_s(Prop),
      opt_semi.
property_s_list([]) --> [].


bindings_map_definition(bindings_map_definition(TypedId, BindL)) -->
      [token(bindings, key)],
      typed_identifier(TypedId),
      [token('=', misc)],
      !,
      startblock,
      bindings_list(BindL),
      endblock.
bindings_map_definition(unnamed_bindings_map_definition(BindL)) -->
      [token(bindings, key)],
      startblock,
      bindings_list(BindL),
      endblock.

bindings_list([Bind|BindL]) -->
      binding(Bind),
      [token(';', misc)],
      bindings_list(BindL).
bindings_list([Bind]) -->
      binding(Bind),
      opt_semi.

binding(Id) -->
      segment_metavar(Id).
binding(binding(BoundNam, BindNam)) -->
      acme_name(BoundNam),
      [token(to, key)],
      acme_name(BindNam).

other_map_definition(other_map_definition(Lang, ExtPars)) -->
      language(Lang),
      [token(':', misc)],
      externally_parsed(ExtPars).

single_predicate(Id) -->
      segment_metavar(Id).
```

```
single_predicate(predicate(Pred, MetaPred)) -->
      actual_predicate(Pred),
      meta_predicate_or_epsilon(MetaPred).

actual_predicate(ExtensDesc) -->
      extension_description(ExtensDesc).
actual_predicate(RawProp) -->
      raw_property(RawProp).


meta_predicate_or_epsilon(MetaPred) -->
      meta_predicate(MetaPred).
meta_predicate_or_epsilon([]) --> [].

meta_predicate(meta_predicate(ConstrL)) -->
      [token('<<', misc)],
      constraints_plus(ConstrL),
      [token('>>', misc)].

constraints_plus([Consts]) -->
      constraints(Consts).
constraints_plus([Consts|ConstrL]) -->
      constraints(Consts),
      [token(';', misc)],
      constraints_plus(ConstrL).

extension_description(extension_description(Lang, ExtPars)) -->
      language(Lang),
      [token(':', misc)],
      externally_parsed(ExtPars).

language(language(Nam)) -->
      acme_name(Nam).

externally_parsed([]) --> [].

raw_property(typed_raw_property(Attr, PropTyp, AttrVal)) -->
      attribute(Attr),
      [token(':', misc)],
      property_type(PropTyp),
      [token('=', misc)],
      !,
      attribute_value(AttrVal).
raw_property(raw_property(Attr, AttrVal)) -->
      attribute(Attr),
      [token('=', misc)],
      !,
      attribute_value(AttrVal).
raw_property(unidentified_raw_property(Attr, PropTyp)) -->
      attribute(Attr),
      [token(':', misc)],
      property_type(PropTyp).

property_type_declaration(property_type_declaration(TypNam, PropTyp)) -->
      [token(property, key)],
      [token(type, key)],
      acme_name(TypNam),
      [token('=', misc)],
      property_type(PropTyp).
property_type_declaration(external_property_type_declaration(TypNam, Lang)) -->
      [token(property, key)],
      [token(type, key)],
      acme_name(TypNam),
      external(Lang).

property_s(Id) -->
      segment_metavar(Id).
property_s(Prop) -->
      property(Prop).
property_s(Props) -->
      properties(Props).

property(property(SingPred)) -->
      [token(property, key)],
      single_predicate(SingPred),
      opt_semi.

properties(properties(SingPredL)) -->
      [token(properties, key)],
      startblock,
      single_predicate_list(SingPredL),
      endblock.

single_predicate_list([SingPred]) -->
      single_predicate(SingPred),
      opt_semi.
single_predicate_list([SingPred|SingPredl]) -->
      single_predicate(SingPred),
      [token(';', misc)],
      single_predicate_list(SingPredl).

property_type(PrimPropTyp) -->
      primitive_property_type(PrimPropTyp).
property_type(PropTypDesc) -->
      property_type_description(PropTypDesc).
property_type(property_type(Nam)) -->
      acme_name(Nam).

external(external_language(Lang)) -->
      [token(external, key)],
      [token(language, key)],
      language(Lang).

attribute(attribute(Nam)) -->
      acme_name(Nam).

attribute_value(Id)   --> segment_metavar(Id).
attribute_value(Str)  --> string(Str).
attribute_value(Num)  --> number(Num).
attribute_value(Nam)  --> acme_name(Nam).
attribute_value(Set)  --> set(Set).
attribute_value(Rec)  --> record(Rec).
attribute_value(Seq)  --> sequence(Seq).
```

```
field_and_value(Id) -->
      segment_metavar(Id).
field_and_value(field_and_value(Nam, AttrVal)) -->
      acme_name(Nam),
      [token('=', misc)],
      attribute_value(AttrVal).

primitive_property_type(primitive_property_type(float)) -->
      [token(float, key)].
primitive_property_type(primitive_property_type(int)) -->
      [token(int, key)].
primitive_property_type(primitive_property_type(char)) -->
      [token(char, key)].
primitive_property_type(primitive_property_type(string)) -->
      [token(string, key)].
primitive_property_type(primitive_property_type(boolean)) -->
      [token(boolean, key)].
primitive_property_type(primitive_property_type(any)) -->
      [token(any, key)].

property_type_description(RecDecl) -->
      record_declaration(RecDecl).
property_type_description(SetDecl) -->
      set_declaration(SetDecl).
property_type_description(SeqDecl) -->
      sequence_declaration(SeqDecl).
property_type_description(EnumDecl) -->
      enum_declaration(EnumDecl).


record_declaration(open_record_declaration(RecParamL)) -->
      [token(open, key)],
      !,
      [token(record, key)],
      [token('[', misc)],
      record_parameter_list(RecParamL),
      [token(']', misc)].
record_declaration(record_declaration(RecParamL)) -->
      [token(record, key)],
      [token('[', misc)],
      record_parameter_list(RecParamL),
      [token(']', misc)].

record_parameter_list([RecParam|RecParamL]) -->
      record_parameter(RecParam),
      [token(';', misc)],
      record_parameter_list(RecParamL).
record_parameter_list([RecParam]) -->
      record_parameter(RecParam),
      opt_semi.

record_parameter(Id) -->
      segment_metavar(Id).
record_parameter(record_parameter(IdML, TypNam)) -->
      id_or_meta_list(IdML),
      [token(':', misc)],
      type_name(TypNam).
```

```
record(record(FieldValL)) -->
      [token('[', misc)],
      field_and_value_list(FieldValL),
      [token(']', misc)].

set_declaration(set_declaration(TypNam)) -->
      [token(set, key)],
      startblock,
      type_name(TypNam),
      endblock.

set(set(AttValL)) -->
      [token('{', misc)],
      attribute_value_list(AttValL),
      [token('}', misc)].

sequence_declaration(sequence_declaration(TypNam)) -->
      [token(sequence, key)],
      [token('<', misc)],
      !,
      type_name(TypNam),
      [token('>', misc)].
sequence_declaration(empty_sequence_declaration) -->
      [token(sequence, key)].

sequence(sequence(AttrValL)) -->
      [token('<', misc)],
      attribute_value_list(AttrValL),
      [token('>', misc)].

enum_declaration(enum_declaration(NamL)) -->
      [token(enum, key)],
      startblock,
      name_plus(NamL),
      endblock.

field_and_value_list([FieldVal]) -->
      field_and_value(FieldVal).
field_and_value_list([FieldVal|FieldValL]) -->
      field_and_value(FieldVal),
      [token(';', misc)],
      field_and_value_list(FieldValL).

attribute_value_list([AttrVal|AttrValL]) -->
      attribute_value(AttrVal),
      [token(',', misc)],
      !,
      attribute_value_list(AttrValL).
attribute_value_list([AttrVal]) -->
      attribute_value(AttrVal).


name_plus([Nam|NamL]) -->
      acme_name(Nam),
      [token(',', misc)],
      name_plus(NamL).
name_plus([Nam]) -->
      acme_name(Nam).

acme_name(Id) -->
      segment_metavar(Id).
acme_name(dotted_acme_name(Id1, Id2)) -->
      id_or_meta(Id1),
      [token('.', misc)],
      !,
      id_or_meta(Id2).
acme_name(uparrowed_acme_name(Id1, Id2)) -->
      id_or_meta(Id1),
      [token('^', misc)],
      !,
      id_or_meta(Id2).
acme_name(IdM) -->
      id_or_meta(IdM).

id_or_meta_list([IdM|IdML]) -->
      id_or_meta(IdM),
      [token(',', misc)],
      id_or_meta_list(IdML).
id_or_meta_list([IdM]) -->
      id_or_meta(IdM).

id_or_meta(Id) -->
      segment_metavar(Id).
id_or_meta(Id) -->
        metavar(Id).
id_or_meta(id(Id)) -->
      acme_identifier(Id).

metavar(conjoined_metavars(Ms)) -->
        [token('@', misc)],
        [token('&', misc)],
        !,
        [token('(', misc)],
        metavar_list(Ms),
        [token(')', misc)].
metavar(generated_metavar(R)) -->
        [token('@', misc)],
        [token('$', misc)],
        !,
        [token('(', misc)],
        acme_identifier(R),
        [token(')', misc)].
metavar(metavar(Id)) -->
      [token('@', misc)],
      acme_identifier(Id).

metavar_list([M|Ms]) -->
     metavar(M),
     [token(',', misc)],
     metavar_list(Ms).
metavar_list([M]) -->
     metavar(M).
```

```
acme_identifier(Id) -->
      [Tok],
      {Tok = token(Id, id)}.

segment_metavar(concatenated_segment_metavars(SMList)) -->
      [token('@', misc)],
      [token('*', misc)],
      !,
      [token('(', misc)],
      segment_metavar_list(SMList),
      [token(')', misc)].
segment_metavar(renamed_segment_metavar(M, SM)) -->
      [token('@', misc)],
      [token('_', misc)],
      !,
      [token('(', misc)],
      metavar(M),
      [token(',', misc)],
      segment_metavar(SM),
      [token(')', misc)].
segment_metavar(dotted_segment_metavar(M, SM)) -->
      [token('@', misc)],
      [token('.', misc)],
      !,
      [token('(', misc)],
      metavar(M),
      [token(',', misc)],
      segment_metavar(SM),
      [token(')', misc)].
segment_metavar(segment_metavar(Id)) -->
      [token('@@', misc)],
      acme_identifier(Id).

segment_metavar_list([SM|SMList]) -->
     segment_metavar(SM),
     [token(',', misc)],
     segment_metavar_list(SMList).
segment_metavar_list([SM]) -->
     segment_metavar(SM).

string(string(Str)) -->
      [Tok],
      {Tok = token(Str, str)}.

number(number(Num)) -->
      [Tok],
      {Tok = token(Num, num)}.

startblock -->
      [token('{', misc)].

endblock -->
      [token('}', misc)],
      opt_semi.

opt_semi --> [token(';', misc)].
opt_semi --> [].
```

3.2 Developing Abstraction Transformations

The tool set delivered includes two useful tools for gauge development that are not actually part of the final gauge itself. (Tools that are part of the gauges will be discussed in the context of the examples, below.)

The parser that is part of the gauges essentially assumes that the descriptions and abstraction rules it deals with are syntactically correct, in the sense that, in case of a syntactic error, it simply fails without providing a useful message indicating the likely location of the error. This is quite common for parsers implemented using the Prolog grammar rule mechanism. In recursive descent parsing, failure of an attempted parse is no reason to suppose that a grammatical error has been discovered, for, most often, backtracking will discover a successful alternative parse.

The tool set includes an enhanced version of the extended Acme parser that keeps track of the farthest it gets in the list of tokens, and reports that location as the likely source of an error when it ultimately fails. While much less efficient than the standard parser, it is very useful in identifying syntactic errors in architecture descriptions and abstraction rules. This version of the parser is in the file `acme-gr-rules-counting.pl`. It is used just like any other Prolog parser.

In gauges, abstraction transformations are applied automatically to update abstract models. Although abstraction transformations are relatively easy to write, some debugging is likely to be necessary. A good first step in developing a stock of abstraction transformations for a gauge is to try applying them "by hand", using the Acme mapping generator (AMG) tool, which is included as part of our DASADA technology distribution. AMG is an interactive tool in which an architectural description, an abstraction rule, and match (i.e., a set of pattern variable bindings) are selected by the user. AMG then generates the abstract description, and adds it to the stock of descriptions available for further transformation. This sort of manual generation of abstract descriptions helps both in eliminating semantic bugs in the abstraction rules and in defining an automatic abstraction strategy for use in the gauge. The source code for AMG can be found in the file `amg.pl`.

Chapter 4

A Dependability Gauge for Failure Tolerance

The first dependability gauge suite we have implemented measures a simple failure-tolerance property of our dynamically reconfigurable version of SEAS. The analysis technology employed is model checking.

4.1 SEAS

SEAS is a component-based system, in that it uses several pre-existing general-purpose AIC software packages as components. Here is a description of the base architecture for this version of SEAS, which is simply a codification of a description that the AIC provides in Acme.[1] While quite simple, considering the size of SEAS, it turns out to be adequate for our purposes, as the failure-tolerance property of interest is specifically concerned with failure of the persistent store.

```
system seas = {
  component cl_http_server = {
    ports {
      http_server_port_for_browser;
      http_server_port_for_html_generator;
    };
    property functions_performed = { web_service };
```

[1]The architecture description used in the demonstration actually has several additional propoerties of components and connectors specified that are relevant to other gauges, but not to the failure-tolerance gauge. In addition, the component representing the client interacting with SEAS has been ignored. These have been omitted for simplicity, and, as a result, the length of the description has been reduced by about one-half.

```
  property allocated_to_machines = { seas_machine };
};
component seas_html_generator = {
  ports {
    html_generator_port_for_server;
    html_generator_port_for_grasper;
    html_generator_port_for_gister;
    html_generator_port_for_gkb_editor;
    html_generator_port_for_ontology_mgr;
  };
  property functions_performed = { html_generation };
  property allocated_to_machines = { seas_machine };
};
component grasper = {
  ports {
    grasper_port_for_html_generator;
    grasper_port_for_gister;
    grasper_port_for_gkb_editor;
  };
  property functions_performed = { summary_graphics_production };
  property allocated_to_machines = { seas_machine };
};
component gister = {
  ports {
    gister_port_for_grasper;
    gister_port_for_html_generator;
  };
  property functions_performed = { answer_calculation };
  property allocated_to_machines = { seas_machine };
};
component generic_knowledge_base_editor = {
  ports {
    gkb_editor_port_for_grasper;
    gkb_editor_port_for_html_generator;
    gkb_editor_port_for_ocelot;
  };
 property functions_performed = { knowledge_base_interface };
 property allocated_to_machines = { seas_machine };
};
component ontology_manager = {
  ports {
    ontology_mgr_port_for_html_generator;
    ontology_mgr_port_for_ocelot;
  };
  property functions_performed = { answer_update };
 property allocated_to_machines = { seas_machine };
};
component ocelot_knowledge_base_management_system = {
  ports {
    ocelot_port_for_gkb_editor;
    ocelot_port_for_ontology_mgr;
    ocelot_port_for_perk;
  };
 property functions_performed = { updates_frame_values };
 property allocated_to_machines = { seas_machine };
};
component perk_storage_system = {
  ports {
    perk_port_for_ocelot;
    perk_port_for_persistent_store;
  };
  property functions_performed = { maintain_persistent_store };
  property allocated_to_machines = { seas_machine };
};
component persistent_store = {
  ports {
    persistent_store_port_for_perk;
  };
  property functions_performed = { persistent_storage };
  property allocated_to_machines = { seas_machine };
};
connector http_server_to_html_generator = {
  roles {
    http_server_role_in_connector_to_html_generator;
    html_generator_role_in_connector_to_http_server;
  };
};
 connector html_generator_to_grasper = {
   roles {
     html_generator_role_in_connector_to_grasper;
     grasper_role_in_connector_to_html_generator;
   };
};
connector html_generator_to_gister = {
  roles {
    html_generator_role_in_connector_to_gister;
    gister_role_in_connector_to_html_generator;
  };
};
connector html_generator_to_gkb_editor = {
  roles {
    html_generator_role_in_connector_to_gkb_editor;
    gkb_editor_role_in_connector_to_html_generator;
  };
};
connector html_generator_to_ontology_mgr = {
  roles {
    html_generator_role_in_connector_to_ontology_mgr;
    ontology_mgr_role_in_connector_to_html_generator;
  };
};
connector grasper_to_gister = {
  roles {
    grasper_role_in_connector_to_gister;
    gister_role_in_connector_to_grasper;
  };
};
connector grasper_to_gkb_editor = {
  roles {
    grasper_role_in_connector_to_gkb_editor;
    gkb_editor_role_in_connector_to_grasper;
  };
};
connector gkb_editor_to_ocelot = {
  roles {
    gkb_editor_role_in_connector_to_ocelot;
    ocelot_role_in_connector_to_gkb_editor;
  };
};
connector ontology_mgr_to_ocelot = {
  roles {
    ontology_mgr_role_in_connector_to_ocelot;
    ocelot_role_in_connector_to_ontology_mgr;
  };
};
connector ocelot_to_perk = {
  roles {
    ocelot_role_in_connector_to_perk;
    perk_role_in_connector_to_ocelot;
  };
};
connector perk_to_persistent_store = {
  roles {
    perk_role_in_connector_to_persistent_store;
    persistent_store_role_in_connector_to_perk;
  };
};
attachments {
  http_server_port_for_html_generator
    to http_server_role_in_connector_to_html_generator;
  html_generator_port_for_http_server
    to html_generator_role_in_connector_to_http_server;
  html_generator_port_for_grasper
    to html_generator_role_in_connector_to_grasper;
  grasper_port_for_html_generator
    to grasper_role_in_connector_to_html_generator;
  html_generator_port_for_gister
    to html_generator_role_in_connector_to_gister;
  gister_port_for_html_generator
    to gister_role_in_connector_to_html_generator;
  html_generator_port_for_gkb_editor
    to html_generator_role_in_connector_to_gkb_editor;
  gkb_editor_port_for_html_generator
    to gkb_editor_role_in_connector_to_html_generator;
  html_generator_port_for_ontology_mgr
    to html_generator_role_in_connector_to_ontology_mgr;
  ontology_mgr_port_for_html_generator
    to ontology_mgr_role_in_connector_to_html_generator;
  grasper_port_for_gister
    to grasper_role_in_connector_to_gister;
  gister_port_for_grasper
    to gister_role_in_connector_to_grasper;
  grasper_port_for_gkb_editor
    to grasper_role_in_connector_to_gkb_editor;
  gkb_editor_port_for_grasper
    to gkb_editor_role_in_connector_to_grasper;
  gkb_editor_port_for_ocelot
    to gkb_editor_role_in_connector_to_ocelot;
  ocelot_port_for_gkb_editor
    to ocelot_role_in_connector_to_gkb_editor;
  ontology_mgr_port_for_ocelot
```

```
      to ontology_mgr_role_in_connector_to_ocelot;
    ocelot_port_for_ontology_mgr
      to ocelot_role_in_connector_to_ontology_mgr;
    ocelot_port_for_perk
      to ocelot_role_in_connector_to_perk;
    perk_port_for_ocelot
      to perk_role_in_connector_to_ocelot;
    perk_port_for_persistent_store
      to perk_role_in_connector_to_persistent_store;
    persistent_store_port_for_perk
      to persistent_store_role_in_connector_to_perk;
  };
}
```

4.2 Abstraction

The basic idea in abstraction is to eliminate all information in the specification that is irrelevant to determining whether the architecture is adequately failure tolerant. In order to do so, we repeatedly apply the two abstraction transformations.[2] The

[2]As noted earlier, some properties of components and connectors irrelevant to this gauge have been omitted from the SEAS architectural description in the interest of brevity. Either before or after applying the bundling transformations, these properties should be stripped off to achieve maximum abstraction. The stripping transformations are gauge specific, and have the form

```
transformation strip_irrelevant_component_property from
  system @s = {
    component @c = {
      property @prop = @val;
      @@rest_c
    };
    @@rest_s
  }
to
  system @s = {
    component @c = {
      @@rest_c
    };
    @@rest_s
  }
provided
  code{
    not member(@prop, gauge-specific list of properties)
  }
```

and

```
transformation strip_irrelevant_connector_property from
  system @s = {
    connector @k = {
      property @prop = @val;
      @@rest_k
    };
    @@rest_s
  }
to
  system @s = {
    component @k = {
      @@rest_k
```

first says that a connected pair of noncritical components can be collapsed to a single component.

```
transformation bundle_noncritical_components from
   system @s = {
     component @c1 = {
       ports {@op; @@ps1};
       property functions_computed = {@@fns1};
       property criticality: int = 0;
     };
     component @c2 = {
       ports {@ip; @@ps2};
       property functions_computed = {@@fns2};
       property criticality: int = @cr;
     };
     connector @k = { roles {@ir; @or}};
     attachments {
       @k.@ir to @c1.@op;
       @k.@or to @c2.@ip;
       @@att
     };
     @@rest
   }
to
   system @s = {
     component @&(@c1, @c2) = {
       ports {@*(@_(@c1,@@ps1), @_(@c2,@@ps2))};
       property functions_computed = {@*(@@fns1, @@fns2)};
       property criticality: int = @cr;
     };
     attachments {
       @@att
     };
     @@rest
   }
where
   map connected_components_to_component with {
```

```
        };
        @@rest_s
      }
    provided
      code{
        not member(@prop, gauge-specific list of properties)
      }
```

The advantage of stripping them prior to bundling is that one need not worry about maintaining the values of these properties when bundling. The advantage of stripping them after bundling is that the bundling rules can be written in a form that is less sensitive to the properties present. Since making the rules as general as possible is crucial in building a stock of reusable abstraction transformations, we have opted for the latter approach. But rather than dealing with the fully general case in discussing this example, a single "irrelevant" property, `functions_computed`, has been left in the description, and the rules have been written to maintain the value of that property when bundling. "Irrelevant" properties in general are handled in much the same way as the example.

```
     [first_component = @c1; first_port = @c1.@op;
          connecting_connector = @k; first_role = @k.@ir; second_role = @k.@or;
          second_component = @c2; second_port = @c2.@ip]
        to @&(@c1, @c2);
     @.(@c1,@@ps1) to @.(@&(@c1,@c2),@_(@c1,@@ps1));
     @.(@c2,@@ps2) to @.(@&(@c1,@c2),@_(@c2,@@ps2));
   }
provided
  // A simple example of the constrint capability: value of @cr can
  // be anything that can be proved to be equal to 0, not just a literal '0'
  code{
    @cr = 0
  }
```

The second transformation used in abstraction says that multiple connectors between a pair of components can be collapsed to a single connector.

```
transformation bundle_connectors from
   system @s = {
     component @c1 = { ports {@op1; @op2; @@ps1; }};
     component @c2 = { ports {@ip1; @ip2; @@ps2; } };
     connector @k1 = { roles {@or1; @ir1; @@rs1; }};
     connector @k2 = { roles {@or2; @ir2; @@rs2; }};
     attachments {
       @k1.@ir1 to @c1.@op1;
       @k1.@or1 to @c2.@ip1;
       @k2.@ir2 to @c1.@op2;
       @k2.@or2 to @c2.@ip2;
       @@att;
     };
     @@rest;
   }
to
   system @s = {
     component @c1 = { ports {@op1; @@ps1; }};
     component @c2 = { ports {@ip1; @@ps2; }};
     connector @k1 = { roles {@or1; @ir1; @@rs1; }};
     attachments {
       @k1.@ir1 to @c1.@op1;
       @k1.@or1 to @c2.@ip1;
       @@att;
        };
     @@rest;
   }
where
   map connectors_to_connector with {
     <@op1, @op2> to @op1;
     <@ip1, @ip2> to @ip1;
     <@k1, @k2> to @k1;
     <@or1, @op2> to @or1;
     <@ir1, @ir2> to @ir1;
   }
```

Clearly, neither of these bundling transformations can produce an abstract architectural description that satisfies the failure-tolerance requirement from a concrete architectural description that does not.[3]

Any sequence of applications of these two abstraction transformations preserves the property of interest. Our goal is to devise a strategy for applying the transformations that results in a sufficiently abstract description for an appropriate variety of architectures.

An effective strategy description must address both the selection of transformations at each step in the abstraction process and the selection of bindings of pattern variables used when matching each transformation against the architecture. Strategies can range from the trivial and inflexible — fixed lists of transformations and bindings — to more sophisticated, loosely specified sequences of applications in which aspects of one transformation application (such as bindings) inform the choices of transformations later on.

In many cases, bindings need not be specified directly. Indeed, "robust" strategies that may be applied to a variety of architectures cannot rely heavily on specific bindings. If we ignore bindings, a straightforward approach to representing simple strategies is to define recursively applicable primitives for repeating or selectively applying transformations or strategies. For example, we can write

$$\sigma_1 \, \sigma_2$$

to represent the strategy "apply strategy σ_1, then apply the strategy σ_2",

$$\sigma^*$$

to represent the strategy of repeatedly applying σ until it is no longer applicable (i.e., until either it can no longer be applied because preconditions for its application are not satisfied or a fixed point has been reached), and

$$[\sigma]$$

to indicate "optional" application of σ (in other words, application of σ if and only if it is applicable). A moment's thought reveals that a strategy such as

$$({T_2}^* \, T_1)^*$$

where T_1 is `bundle_noncritical_components` and T_2 is `bundle_connectors`, is sufficient to generate a "maximally abstract" — relative to the failure-tolerance property of interest — representation of our example system:

[3]The co-preservation of failure tolerance could be formally verified by proving that the transformations always produce an abstract architecture whose theory is faithfully interpretable in the theory of the concrete architecture [9, 14, 15].

```
system seas = {
  component seas_main = {
    ports {
      seas_main_port_for_browser;
      seas_main_port_for_persistent_store;
    };
    property functions_performed = {
      web_service,
      html_generation,
      summary_graphics_production,
      answer_calculation,
      knowledge_base_interface,
      answer_update,
      updates_frame_values,
      maintain_persistent_store
    };
    property allocated_to_machines = { seas_machine };
  };
  component persistent_store = {
    ports {
      persistent_store_port_for_seas_main;
    };
    property functions_performed = { persistent_storage };
    property allocated_to_machines = { seas_machine };
  };
  connector seas_main_to_persistent_store = {
    roles {
      seas_main_role_in_connector_to_persistent_store;
      persistent_store_role_in_connector_to_seas_main; };
  };
  attachments {
    seas_main_port_for_persistent_store
      to seas_main_role_in_connector_to_persistent_store;
    persistent_store_port_for_seas_main
      to persistent_store_role_in_connector_to_seas_main;
  };
}
```

4.3 Analysis

In the analysis phase of the gauge, the failure-tolerance property and the abstract Acme description are “translated” into Promela, the input language of the Spin model checker. The result of this translation is

```
bit compromised_seas_main = 0;         /* Initially, no intrusion */
bit compromised_persistent_store = 0;  /* Initially, no intrusion */
int attack_attempts = 0;

proctype attack_seas(){
reattack:
  if
  :: !compromised_persistent_store -->
     reattack_persistent_store:
       if
       :: attack_attempts++;
          printf("It took %d attacks to compromise persistent_store.\\n",
                 attack_attempts);             /* Record before failure state */
          compromised_persistent_store  = 1;   /* Attack succeeds */
          goto reattack
       :: attack_attempts++;
          goto reattack_persistent_store      /* Attack fails */
       fi
  :: !compromised_seas_main -->
     reattack_seas_main:
       if
       :: attack_attempts++;
          printf("It took %d attacks to compromise seas_main.\\n",
                 attack_attempts);
          compromised_seas_main  = 1;      /* Attack succeeds */
          goto reattack
       :: attack_attempts++;
          goto reattack_seas_main          /* Attack fails */
       fi
  :: else --> skip
fi;
printf("It took %d attack attempts to compromise all components.\\n",
        attack_attempts)
};

init{
  run attack_seas()
}
```

which, when analyzed by Spin, determines that a single successful attack on the right component can result in system failure.

4.4 Update

An obvious way to increase the level of failure tolerance is to make the critical component redundant (using diverse implementations, with diverse vulnerabilities, if intrusion tolerance as well as fault tolerance is required). Toggling the SEAS reconfiguration switch that duplicates and distributes the redundant storage component results in a new concrete architecture:

```
system seas = {
  component cl_http_server = {
    ports {
      http_server_port_for_browser;
      http_server_port_for_html_generator;
    };
    property functions_performed = { web_service };
    property allocated_to_machines = { seas_machine };
  };
  component seas_html_generator = {
```

```
  ports {
    html_generator_port_for_server;
    html_generator_port_for_grasper;
    html_generator_port_for_gister;
    html_generator_port_for_gkb_editor;
    html_generator_port_for_ontology_mgr;
  };
  property functions_performed = { html_generation };
  property allocated_to_machines = { seas_machine };
};
component grasper = {
  ports {
    grasper_port_for_html_generator;
    grasper_port_for_gister;
    grasper_port_for_gkb_editor;
  };
  property functions_performed = { summary_graphics_production };
  property allocated_to_machines = { seas_machine };
};
component gister = {
  ports {
    gister_port_for_grasper;
    gister_port_for_html_generator;
  };
  property functions_performed = { answer_calculation };
  property allocated_to_machines = { seas_machine };
};
component generic_knowledge_base_editor = {
  ports {
    gkb_editor_port_for_grasper;
    gkb_editor_port_for_html_generator;
    gkb_editor_port_for_ocelot;
  };
 property functions_performed = { knowledge_base_interface };
 property allocated_to_machines = { seas_machine };
};
component ontology_manager = {
  ports {
    ontology_mgr_port_for_html_generator;
    ontology_mgr_port_for_ocelot;
  };
  property functions_performed = { answer_update };
 property allocated_to_machines = { seas_machine };
};
component ocelot_knowledge_base_management_system = {
  ports {
    ocelot_port_for_gkb_editor;
    ocelot_port_for_ontology_mgr;
    ocelot_port_for_perk;
  };
 property functions_performed = { updates_frame_values };
 property allocated_to_machines = { seas_machine };
};
component perk_storage_system = {
  ports {
    perk_port_for_ocelot;
    perk_port_for_persistent_store_1;
    perk_port_for_persistent_store_2;
    perk_port_for_persistent_store_3;
  };
  property functions_performed = { maintain_persistent_store };
  property allocated_to_machines = { seas_machine };
};
component persistent_store_1 = {
  ports {
    persistent_store_1_port_for_perk;
  };
  property functions_performed = { persistent_storage };
  property allocated_to_machines = { storage_machine_1 };
};
component persistent_store_2 = {
  ports {
    persistent_store_2_port_for_perk;
  };
  property functions_performed = { persistent_storage };
  property allocated_to_machines = { storage_machine_2 };
};
component persistent_store_3 = {
  ports {
    persistent_store_3_port_for_perk;
  };
  property functions_performed = { persistent_storage };
  property allocated_to_machines = { storage_machine_3 };
};
connector http_server_to_html_generator = {
  roles {
    http_server_role_in_connector_to_html_generator;
    html_generator_role_in_connector_to_http_server;
  };
};
 connector html_generator_to_grasper = {
   roles {
     html_generator_role_in_connector_to_grasper;
     grasper_role_in_connector_to_html_generator;
   };
};
connector html_generator_to_gister = {
  roles {
    html_generator_role_in_connector_to_gister;
    gister_role_in_connector_to_html_generator;
  };
};
connector html_generator_to_gkb_editor = {
  roles {
    html_generator_role_in_connector_to_gkb_editor;
    gkb_editor_role_in_connector_to_html_generator;
  };
};
connector html_generator_to_ontology_mgr = {
  roles {
    html_generator_role_in_connector_to_ontology_mgr;
    ontology_mgr_role_in_connector_to_html_generator;
  };
};
```

```
connector grasper_to_gister = {
  roles {
    grasper_role_in_connector_to_gister;
    gister_role_in_connector_to_grasper;
  };
};
connector grasper_to_gkb_editor = {
  roles {
    grasper_role_in_connector_to_gkb_editor;
    gkb_editor_role_in_connector_to_grasper;
  };
};
connector gkb_editor_to_ocelot = {
  roles {
    gkb_editor_role_in_connector_to_ocelot;
    ocelot_role_in_connector_to_gkb_editor;
  };
};
connector ontology_mgr_to_ocelot = {
  roles {
    ontology_mgr_role_in_connector_to_ocelot;
    ocelot_role_in_connector_to_ontology_mgr;
  };
};
connector ocelot_to_perk = {
  roles {
    ocelot_role_in_connector_to_perk;
    perk_role_in_connector_to_ocelot;
  };
};
connector perk_to_persistent_store_1 = {
  roles {
    perk_role_in_connector_to_persistent_store_1;
    persistent_store_1_role_in_connector_to_perk;
  };
};
connector perk_to_persistent_store_2 = {
  roles {
    perk_role_in_connector_to_persistent_store_2;
    persistent_store_2_role_in_connector_to_perk;
  };
};
connector perk_to_persistent_store_3 = {
  roles {
    perk_role_in_connector_to_persistent_store_3;
    persistent_store_3_role_in_connector_to_perk;
  };
};
attachments {
  http_server_port_for_html_generator
    to http_server_role_in_connector_to_html_generator;
  html_generator_port_for_http_server
    to html_generator_role_in_connector_to_http_server;
  html_generator_port_for_grasper
    to html_generator_role_in_connector_to_grasper;
  grasper_port_for_html_generator
    to grasper_role_in_connector_to_html_generator;
    html_generator_port_for_gister
      to html_generator_role_in_connector_to_gister;
    gister_port_for_html_generator
      to gister_role_in_connector_to_html_generator;
    html_generator_port_for_gkb_editor
      to html_generator_role_in_connector_to_gkb_editor;
    gkb_editor_port_for_html_generator
      to gkb_editor_role_in_connector_to_html_generator;
    html_generator_port_for_ontology_mgr
      to html_generator_role_in_connector_to_ontology_mgr;
    ontology_mgr_port_for_html_generator
      to ontology_mgr_role_in_connector_to_html_generator;
    grasper_port_for_gister
      to grasper_role_in_connector_to_gister;
    gister_port_for_grasper
      to gister_role_in_connector_to_grasper;
    grasper_port_for_gkb_editor
      to grasper_role_in_connector_to_gkb_editor;
    gkb_editor_port_for_grasper
      to gkb_editor_role_in_connector_to_grasper;
    gkb_editor_port_for_ocelot
      to gkb_editor_role_in_connector_to_ocelot;
    ocelot_port_for_gkb_editor
      to ocelot_role_in_connector_to_gkb_editor;
    ontology_mgr_port_for_ocelot
      to ontology_mgr_role_in_connector_to_ocelot;
    ocelot_port_for_ontology_mgr
      to ocelot_role_in_connector_to_ontology_mgr;
    ocelot_port_for_perk
      to ocelot_role_in_connector_to_perk;
    perk_port_for_ocelot
      to perk_role_in_connector_to_ocelot;
    perk_port_for_persistent_store_1
      to perk_role_in_connector_to_persistent_store_1;
    persistent_store_port_1_for_perk
      to persistent_store_1_role_in_connector_to_perk;
    perk_port_for_persistent_store_2
      to perk_role_in_connector_to_persistent_store_2;
    persistent_store_port_2_for_perk
      to persistent_store_2_role_in_connector_to_perk;
    perk_port_for_persistent_store_3
      to perk_role_in_connector_to_persistent_store_3;
    persistent_store_port_3_for_perk
      to persistent_store_3_role_in_connector_to_perk;
  };
}
```

Automatically applying the same two abstraction rules in accordance with the same strategy produces an updated abstract description:

```
system seas = {
  component seas_main = {
    ports {
      seas_main_port_for_browser;
      seas_main_port_for_persistent_store;
    };
    property functions_performed = {
      web_service,
      html_generation,
      summary_graphics_production,
      answer_calculation,
      knowledge_base_interface,
      answer_update,
      updates_frame_values,
      maintain_persistent_store
    };
    property allocated_to_machines = { seas_machine };
  };
  component persistent_store_1 = {
    ports {
      persistent_store_1_port_for_seas_main;
    };
    property functions_performed = { persistent_storage };
    property allocated_to_machines = { storage_machine_1 };
  };
  component persistent_store_2 = {
    ports {
      persistent_store_2_port_for_seas_main;
    };
    property functions_performed = { persistent_storage };
    property allocated_to_machines = { storage_machine_2 };
  };
  component persistent_store_3 = {
    ports {
      persistent_store_3_port_for_seas_main;
    };
    property functions_performed = { persistent_storage };
    property allocated_to_machines = { storage_machine_3 };
  };
  connector seas_main_to_persistent_store_1 = {
    roles {
      seas_main_role_in_connector_to_persistent_store_1;
      persistent_store_1_role_in_connector_to_seas_main; };
  };
  connector seas_main_to_persistent_store_2 = {
    roles {
      seas_main_role_in_connector_to_persistent_store_2;
      persistent_store_2_role_in_connector_to_seas_main; };
  };
  connector seas_main_to_persistent_store_3 = {
    roles {
      seas_main_role_in_connector_to_persistent_store_3;
      persistent_store_3_role_in_connector_to_seas_main; };
  };
  attachments {
    seas_main_port_for_persistent_store_1
      to seas_main_role_in_connector_to_persistent_store_1;
    persistent_store_1_port_for_seas_main
      to persistent_store_1_role_in_connector_to_seas_main;
    seas_main_port_for_persistent_store_2
      to seas_main_role_in_connector_to_persistent_store_2;
    persistent_store_2_port_for_seas_main
      to persistent_store_2_role_in_connector_to_seas_main;
    seas_main_port_for_persistent_store_3
      to seas_main_role_in_connector_to_persistent_store_3;
    persistent_store_3_port_for_seas_main
      to persistent_store_3_role_in_connector_to_seas_main;
  };
}
```

Automatically translating this description to Promela yields

```
bit compromised_seas_main = 0;           /* Initially, no intrusion */
bit compromised_persistent_store_1 = 0;  /* Initially, no intrusion */
bit compromised_persistent_store_2 = 0;  /* Initially, no intrusion */
bit compromised_persistent_store_3 = 0;  /* Initially, no intrusion */
int attack_attempts = 0;

proctype attack_seas(){
reattack:
  if
  :: !compromised_persistent_store_1 -->
     reattack_persistent_store_1:
       if
       :: attack_attempts++;
          printf("It took %d attacks to compromise persistent_store_1.\\n",
                 attack_attempts);               /* Record before failure state */
          compromised_persistent_store_1  = 1;   /* Attack succeeds */
          goto reattack
       :: attack_attempts++;
          goto reattack_persistent_store_1       /* Attack fails */
       fi
  :: !compromised_persistent_store_2 -->
     reattack_persistent_store_2:
       if
       :: attack_attempts++;
          printf("It took %d attacks to compromise persistent_store_2.\\n",
                 attack_attempts);               /* Record before failure state */
          compromised_persistent_store_2  = 1;   /* Attack succeeds */
          goto reattack
       :: attack_attempts++;
          goto reattack_persistent_store_2       /* Attack fails */
       fi
  :: !compromised_persistent_store_3 -->
     reattack_persistent_store_3:
       if
       :: attack_attempts++;
          printf("It took %d attacks to compromise persistent_store_3.\\n",
                 attack_attempts);               /* Record before failure state */
          compromised_persistent_store_3  = 1;   /* Attack succeeds */
          goto reattack
       :: attack_attempts++;
          goto reattack_persistent_store_3       /* Attack fails */
       fi
  :: !compromised_seas_main -->
     reattack_seas_main:
       if
       :: attack_attempts++;
          printf("It took %d attacks to compromise seas_main.\\n",
                 attack_attempts);
          compromised_seas_main  = 1;      /* Attack succeeds */
          goto reattack
       :: attack_attempts++;
          goto reattack_seas_main          /* Attack fails */
       fi
  :: else --> skip
fi;
printf("It took %d attack attempts to compromise all components.\\n",
        attack_attempts)
};

init{
  run attack_seas()
}
```

and an updated Spin analysis, which shows that failure of a single persistent store component is now tolerated, is used to produce an updated gauge reading.

Note that the strategy employed seems to be fairly robust. The choice of bindings for each transformation is not specified, since the most abstract result is independent of the order in which the bindings are chosen. Furthermore, the same strategy works after the addition and deletion of components or connectors from the noncritical part of the architecture. More specifically, the feasibility of automatically updating gauges has been demonstrated by integrating additional components that redundantly compute a second critical function into the system, resulting in changes to the requirements, functionality, and architecture of the system.

While this robustness is encouraging, it is not surprising to find that there are situations in which this simple approach to encoding strategy is not powerful enough to tackle more complex abstraction problems, and that it does not exhibit sufficient robustness in the face of larger architectural changes. The most general concern is search strategy; a sequence of transformations may result in a “dead end” in which a description that is not sufficiently abstract cannot be transformed further given the set of known atomic transformations. An example would be a strategy that applies T_1 too many times, resulting in edge loops that cannot be abstracted away using T_1 or T_2. More general tree-based search strategies may be required in some cases. Even without complex search strategies, it is useful for earlier transformations to collect information for use by later transformations. In particular, this can help efficiency, since transformations can share replicated work, such as checking constraints. Many transformations, for instance, have only local effects, and knowledge of where a previous transformation was applied can inform the next transformation. Finally, the ability to express more sophisticated constraints may be useful. The current scenario relies on pattern variables to express constraints, meaning the specification pattern must extract information needed for constraint checking. It would be possible to achieve greater separation of information extraction from the pattern replacement process. Examining the degree to which this aids the expression of abstraction strategies is an interesting direction of research.

Chapter 5

A Dependability Gauge for Confidentiality

The second dependability gauge that we have implemented measures a simple security property of the SEAS architecture, whether information that is transmitted between different machines is encoded in a form that prevents eavesdropping by unauthorized parties. In this case, automatic theorem proving is used to determine the gauge reading.

5.1 Abstraction

The approach to abstraction is much the same as for the failure-tolerance gauge. As before, the main abstraction techniques employed are bundling of components and connectors.[1] The only real difference is when to, and when not to, bundle.

- Components should be bundled if and only if they are allocated to the same machine.
- Connectors should be bundled if and only if the ports to which they are attached are ports of components allocated to the same machine.

Modifying the earlier transformations to reflect these changes is completely straightforward, and use of the same abstraction strategy — bundle as many con-

[1] And, as before, the elimination of "irrelevant" properties. Of course, the list of relevant properties is now quite different. For example, `function_performed` is now "irrelevant", while `allocated_to_machine` has become quite relevant.

nectors as possible, then bundle two components (if possible), and repeat until nothing more can be done — again produces an analyzable abstract strategy.

The architectural description that results from this abstraction process is

```
system seas = {
  component seas_client = {
    ports {
      seas_client_port_for_server: html ;
    };
    property allocated_to_machines = { client_machine };
  };
  component seas_server = {
    ports {
      seas_server_port_for_client: html ;
      seas_server_port_for_store: rsa ;
    };
    property allocated_to_machines = { seas_machine };
  };
  component seas_store = {
    ports {
      seas_store_port_for_server: rsa ;
    };
    property allocated_to_machines = { store_machine };
  };
  connector seas_client_to_seas_server = {
    roles {
      seas_client_role_in_client_server_connector: html ;
      seas_server_role_in_client_server_connector: html ;
    };
    property connection_protocol = https ;
  };
  connector seas_server_to_seas_store = {
    roles {
      seas_server_role_in_server_store_connector: rsa ;
      seas_store_role_in_server_store_connector: rsa ;
    };
    property connection_protocol = http ;
  };
  attachments {
    seas_client_port_for_server
      to seas_client_role_in_client_server_connector;
    seas_server_port_for_client
      to seas_server_role_in_client_server_connector;
    seas_server_port_for_store
      to seas_server_role_in_server_store_connector;
    seas_store_port_for_server
      to seas_store_role_in_server_store_connector;
  };
};
```

5.2 Analysis

5.2.1 Reasoning about Encryption

A simple logical theory is adequate for the required reasoning about the effects of encryption. However, the goal is to represent the reasoning in a way that makes the essential concepts explicit, so that the basic framework can be naturally extended to capture more interesting arguments. In other words, the logical theory will not be specially tailored to fit the simple problem under consideration.

The basic idea of the argument of interest is that messages convey content, where content is the sort of thing that agents know, but that content must be extracted from the message (which can be thought of as a string of characters). In some cases, the method for extraction is generally, or at least widely, known, and can be recognized given the message. (Think of natural language text sent in-the-clear.) In other cases, the method for extraction is known only to a select few, and the extraction method cannot be recognized given the message alone. The benefit of encryption is that extraction of content from an encrypted message requires use of just such an extracter.

If agent a knows how to extract the content p of message m from p, then a must know which extraction method g is used to produce p from m, or, equivalently, that g is the inverse of whatever representation method f was used to represent p as m.[2] But, for some choices of f, it is reasonable to assume that no one other than authorized parties — including the sender and intended recipient of m — knows either that m is the result of applying f to p or which g can be

[2]Some subtleties are being deliberately ignored here. "Agent a knows that g' is the extraction method for m" does not follow from "a knows that g is the extraction method for m" and "$g = g'$". The problem is that a may not know that g is g', and may even not recognize that g' is the extraction method for m when directly presented with g'. This problem can be safely ignored for the present because that we are primarily concerned with "knowing how", rather than "knowing that", and what we mean by saying "a knows that g is the extraction method for m" is that, when presented with m *in the standard way*, a can retrieve an extraction method *in the standard way* — a method known to *us* as "g" — which can be used to extract the content of m — which is known to *us* as "p". For present purposes, it is adequate to suppose that, for every pair of terms that denote the same object,

1. everyone who knows one also knows the other, and
2. it is common knowledge among them that the two terms denote the same object, and which object it is,

and that quantification is to be interpreted substitutionally. Given these assumptions, the loose talk about "knowing that" is harmless.

used to obtain p from m. For such f, we can thus infer that no one other than the authorized parties knows that the content of m is p. In other words, assuming limited knowledge of the representation and extraction methods — as we do when encryption is employed — the impossibility of a violation of confidentiality can be demonstrated quite trivially.

Thus, the essence of encryption is simply that it prevents extraction of content by unauthorized agents. That a representation method consisting of application of some particular encryption function to text in some particular language prevents extraction of content by unauthorized agents is, for our purposes, simply an empirical hypothesis. No attempt will be made to show that any encryption scheme actually guarantees that unauthorized extraction is impossible.

Our theory for reasoning about the effects of encryption will be formalized in a multi-sorted first-order logic, with the following sorts:

Event Events are sendings and receivings of messages by agents. Variables x, y, and z (possibly decorated with superscripts or subscripts) will range over Event.

Agent Agents are the possessors of epistemic state. Variables a, b, and c will range over Agent.

Message Messages are sent and received represented contents. The variable m will range over messages.

Content Contents are what messages represent. They can also be thought of as what agents know or fail to know, although the formalization will not reflect this fact (for technical reasons). The variables p and q will range over contents.

Representer Representers produce messages from contents, i.e., they are partial functions from contents to messages. The variable f will range over representers.

Extracter Extracters produce contents from messages, i.e., they are partial functions from messages to contents. The variable g will range over extracters.

Extracter g will be called the *inverse* of f when

$$\forall p \,.\, f(p)\downarrow \Rightarrow g(f(p))\downarrow \wedge\, g(f(p)) = p$$

and

$$\forall m \,.\, g(m)\!\downarrow\, \Rightarrow f(g(m))\!\downarrow \wedge\, f(g(m)) = m$$

i.e., when it is the inverse in the usual sense for partial functions. Since we assume that every representer has an extracter as its inverse, and the uniqueness of this extracter is guaranteed by the definition of "inverse", that g is the inverse of f can be expressed by

$$g = \text{inverse}(f)$$

Since $f(p)\!\downarrow$ follows from $f(p) = m$, the definition of "inverse" licenses the inclusion of the following pair of axioms in our theory:

$$\forall f\, \forall p\, \forall m \,.\, f(p) = m \Rightarrow \text{inverse}(f)(m) = p$$
$$\forall f\, \forall p\, \forall m \,.\, \text{inverse}(f)(m) = p \Rightarrow f(p) = m$$

(Note that the parentheses are being used in two different senses in the formula above. In the expression "$f(p)$", for example, "f" is an individual variable, and "$\cdot(\cdot)$" denotes a function on individuals. In the expression "inverse(f)", "inverse" denotes a function on individuals, and the parentheses indicate application of that function to the individual denoted by "f". So, appearances to the contrary notwithstanding, both the axioms are first-order.)

The fact that every event is either a send or a receive, but never both, can be expressed[3]

$$\forall x \,.\, \text{Send}(x) + \text{Receive}(x)$$

The sender or receiver of an event, and the message that is sent or received, can be determined from the event:

$$\forall x \,.\, \text{who}(x)\!\downarrow \wedge\, \text{body}(x\!\downarrow)$$

For our purposes, agents' knowledge is restricted to having one of three forms. First, that agent a knows that message m has content p is expressed

$$\text{Know}_1(a, m, p)$$

Second, that agent a knows that message m was obtained by applying representer f to some content is expressed

$$\text{Know}_2(a, m, f)$$

[3]The symbol $+$ means "exclusive or".

Third, that agent a knows that the content of message m can be extracted using extracter g can be expressed

$$\text{Know}_3(a, m, g)$$

(Note that dropping the subscript on Know cannot result in ambiguity, given the difference in the sorts of the third arguments.) The following abbreviation schemes seem to help the intuition, at some cost in brevity.

$$\begin{array}{lcl} \text{Know}(a, \text{“}m \text{ says } p\text{”}) & \longrightarrow & \text{Know}_1(a, m, p) \\ \text{Know}(a, \text{“}m \text{ from } f\text{”}) & \longrightarrow & \text{Know}_2(a, m, f) \\ \text{Know}(a, \text{“}g \text{ for } m\text{”}) & \longrightarrow & \text{Know}_3(a, m, g) \end{array}$$

Now, for the argument itself. Let x_0 be the sending of a message m_0 by a_0, where m_0 is $f_0(p_0)$ for some representer f_0, i.e.,

$$\text{Send}(x_0) \wedge \text{who}(x_0) = a_0 \wedge \text{body}(x_0) = m_0 \wedge m_0 = f_0(p_0)$$

We assume that only authorized agents know that inverse(f_0) is required to extract p_0 from m_0, i.e., that

$$\forall b \,.\, \text{Know}(b, \text{“inverse}(f_0) \text{ for } m_0\text{”}) \Rightarrow \text{Auth}(b)$$

We want to be able to show that an unauthorized agent cannot know that the content of m_0 is p_0, i.e.,

$$\forall b \,.\, \text{Know}(b, \text{“}m_0 \text{ says } p_0\text{”}) \Rightarrow \text{Auth}(b)$$

The desired result follows immediately given the general principle that an agent knows that a message has a particular content only if he (or she) knows which extracter to use to extract the content of the message, i.e.,

$$\forall b \, \forall m \, \forall p \, \forall g \,.\, \text{Know}(b, \text{“}m \text{ says } p\text{”}) \wedge g(m) = p \Rightarrow \text{Know}(b, \text{“}g \text{ for } m\text{”})$$

This informal argument can easily be turned into a formal derivation, as in Figure 5.1 (which has been a bit simplified by exclusion of hypotheses that are not needed). This derivation shows that all necessary axioms and hypotheses have been captured, and suggests that any reasonable automatic theorem prover ought to be able to discover a proof.

$\{1\}$	1.	$f_0(p_0) = m_0$	Prem
$\{2\}$	2.	$\forall b \,.\, \text{Know}(b, \text{“inverse}(f_0) \text{ for } m_0\text{”}) \Rightarrow \text{Auth}(b)$	Prem
$\{3\}$	3.	$\forall b\, \forall m\, \forall p\, \forall g \,.\, \text{Know}(b, \text{“}m \text{ says } p\text{”}) \wedge g(m) = p \Rightarrow \text{Know}(b, \text{“}g \text{ for } m\text{”})$	Ax
$\{4\}$	4.	$\forall f\, \forall p\, \forall m \,.\, f(p) = m \Rightarrow \text{inverse}(f)(m) = p$	Ax
$\{5\}$	5.	$\text{Know}(b, \text{“}m_0 \text{ says } p_0\text{”})$	Hyp
$\{4\}$	6.	$f_0(p_0) = m_0 \Rightarrow \text{inverse}(f_0)(m_0) = p_0$	UI (4)
$\{1, 4\}$	7.	$\text{inverse}(f_0)(m_0) = p_0$	MP(6,1)
$\{1, 4, 5\}$	8.	$\text{Know}(b, \text{“}m_0 \text{ says } p_0\text{”}) \wedge \text{inverse}(f_0)(m_0) = p_0$	Conj(5,7)
$\{3\}$	9.	$\text{Know}(b, \text{“}m_0 \text{ says } p_0\text{”}) \wedge \text{inverse}(f_0)(m_0) = p_0 \Rightarrow \text{Know}(b, \text{“inverse}(f_0) \text{ for } m_0\text{”})$	UI(3)
$\{1, 3, 4, 5\}$	10.	$\text{Know}(b, \text{“inverse}(f_0) \text{ for } m_0\text{”})$	MP(9,8)
$\{2\}$	11.	$\text{Know}(b, \text{“inverse}(f_0) \text{ for } m_0\text{”}) \Rightarrow \text{Auth}(b)$	UI(2)
$\{1, 2, 3, 4, 5\}$	12.	$\text{Auth}(b)$	MP(11,10)
$\{1, 2, 3, 4\}$	13.	$\text{Know}(b, \text{“}m_0 \text{ says } p_0\text{”}) \Rightarrow \text{Auth}(b)$	Cond(12,5)
$\{1, 2, 3, 4\}$	13.	$\forall b \,.\, \text{Know}(b, \text{“}m_0 \text{ says } p_0\text{”}) \Rightarrow \text{Auth}(b)$	UG(13)

Figure 5.1: Formal derivation in a representative natural deduction calculus

5.2.2 Analysis of Intramachine Connections

Given the argument in the previous section, the approach to analyzing the confidentiality of intramachine connections in an architecture should be fairly straightforward. The architecture description should include information about an encryption of data prior to transmission in the type(s) of the relevant port(s), and information about the protocol used by the connector as a property of the connector. Given this information, the extracter(s) for messages obtained by eavesdropping on connector traffic can be identified. If that extracter can be assumed to be known only by authorized agents, then the content of the message is secure (and, if not, then not).

In the SEAS example, two different mechanisms are employed to secure the intramachine connections. The client-server connection uses the HTTPS protocol instead of standard HTTP, while the server-store connection relies on the fact that the data is encrypted by RSA prior to storage (and stored in encrypted form, so that compromise of the storage machine cannot contaminate the SEAS database). Thus, two proofs that encryption will provide confidentiality are required. The "translation" of the abstract architectural model to PTTP input reflects this fact.

```
encryption_proof_0 :-
   nl,
   pttp((
          apply(https_representer, p0, m0),
          apply(html_representer, p0, m0),
          (auth(B) ; not_know3(B, m0, inverse(https_representer))),
          (auth(B) ; not_know3(B, m0, inverse(rsa_representer))),
          (know3(B, M, G) ; not_know1(B, M, P) ; not_apply(G, M, P)),
          (apply(inverse(F), M, P) ; not_apply(F, P, M)),
          know1(b, m0, p0),
          (query :- auth(b))
       )),
   fail.
encryption_proof_0 :-
   prove(query).

encryption_proof_1 :-
   nl,
   pttp((
          apply(http_representer, p0, m0),
          apply(rsa_representer, p0, m0),
          (auth(B) ; not_know3(B, m0, inverse(https_representer))),
          (auth(B) ; not_know3(B, m0, inverse(rsa_representer))),
          (know3(B, M, G) ; not_know1(B, M, P) ; not_apply(G, M, P)),
          (apply(inverse(F), M, P) ; not_apply(F, P, M)),
          know1(b, m0, p0),
          (query :- auth(b))
       )),
   fail.
encryption_proof_1 :-
   prove(query).

encryption_proof :-
   dolist([
            encryption_proof_1,
            encryption_proof_0
         ]).
```

PTTP successfully discovers the desired proofs, as expected.

5.3 Update

Just as in the case of the failure-tolerance gauge, the abstraction strategy employed proves to be sufficiently robust that it will successfully produce an updated abstract model after any of the changes to the architecture that can be made to our dynamically reconfigurable SEAS. In particular, if the redundant persistent storage switch is toggled, the abstract description is successfully updated and the four required proofs are successfully discovered.

Chapter 6

Related Work

Transformational implementation was an outgrowth of earlier work on program synthesis. The basic idea is to formalize the process of refining a high-level program specification into executable code. Many of the seminal papers in the field have appeared in anthologies [1, 7, 13]. Experience showed that refinement was a knowledge-intensive process, and the key to success was to focus on a relatively narrow domain. Software architecture is one such domain. At SRI, we have worked on formalizing the process of architecture refinement since 1992 [3, 5, 9, 10, 11, 14, 15, 16]. Recently, we have focused on adapting our technology to component-based systems [17]. Although some other researchers have investigated the notion of mappings between architectural descriptions at different levels of abstraction [8], none, to the best of our knowledge, has attempted to formalize the process of generating the mappings by using transformations that are "correctness preserving". Conversely, most researchers investigating the notion of correct refinement have not focused on refinement of architecture. There are a few exceptions (e.g., [2]). Of particular note is the work of Philipps and Rumpe at T. U. Munich [12], who explicitly address the problem of correct architecture refinement and that of Saridakis and Issarny of IRISA [20] on developing dependable architectures by refinement. Neither employs a transformational framework, however.

Abstract interpretation is the general framework for the definition of abstractions of programs. It consists of a mapping between a concrete and an abstract domain that sends sets of concrete states to single abstract states, together with a mapping from the basic operations or functions of the concrete system to functions of the abstract system. While abstract interpretation is the basis for static analysis techniques used in compilers, it is not widely used for dependability analysis.

It is in fact extremely difficult to construct useful and accurate abstractions that automatically preserve the desired dependability properties. Abstract models are usually provided manually, and theorem proving is used to check that the abstraction mapping preserves the properties. Once the preservation property is established using theorem proving, the abstract model is analyzed by model checking. Recently [4, 18], novel techniques for automatic Boolean abstraction have been developed by SRI. These techniques enable verification of system temporal properties of infinite state systems without manual construction of an abstraction.

Since the abstraction process introduces loss of information by collapsing concrete states into a single abstract state, false negative results may emerge. For instance, a model checker may exhibit an error trace that corresponds to an execution of the abstract program that violates the dependability properties. However, this error trace may not correspond to an execution trace in the concrete program. This situation indicates that the abstraction is too coarse. That is, too many details were abstracted away, and the abstraction needs to be refined. Techniques have been developed recently at SRI [19] to use the error trace to automatically refine the abstraction. The verification methodology — abstraction followed by successive refinements based on the results of model checking — was successfully used to prove safety properties of several systems, including a data link protocol used by Philips Corporation in one of its commercial products. The original proof of the protocol required two months of work and was entirely done using a theorem prover. A Boolean abstraction of the protocol can be automatically generated using the predicates appearing in the description of the protocol in about a hundred seconds with SRI's PVS theorem prover. The abstract protocol is then analyzed in a few seconds to check that all the safety properties hold.

Chapter 7

Conclusion

Dependability gauges provide a technology to monitor evolving dependability properties of dynamically evolving systems. Our approach to building dependability gauages applies proven technology for design time dependability analysis at runtime. The principal innovation consists of focusing on abstraction rather than refinement, and on automatic updating of abstractions and analyses developed at design time after making small, “well structured” changes to architectural requirements and the system architecture. Our emphasis in the future will be to develop a suite of abstraction transformations capable of generating a wide range of dependability gauges, and on developing and experimenting with technologies to make abstraction and analysis more robust.

Our dependability gauge technology is complementary to the more fine-grained runtime analysis that can be performed by monitoring events at component interfaces and within connectors — that is, our technology is complementary to that developed by the other DASADA contractors. For instance, our failure-tolerance gauge provides an excellent example of the potential synergy. Runtime event monitoring of components’ interface behavior can detect some instances of component failure. If failure of a component computing a critical function is observed, and the failure-tolerance gauge for that function shows that only a single component failure can be tolerated, the system, although functioning correctly, is on the verge of failure, and immediate corrective action may be needed. If, on the other hand, the failure-tolerance gauge shows that multiple component failures can be tolerated, the need for corrective action is less urgent. Thus, the combination of component runtime behavior gauges and dependability gauges provides valuable system status information that cannot be obtained with either technology alone.

Bibliography

[1] W. W. Agresti, editor, *Tutorial: New Paradigms for Software Development*, IEEE Computer Society, 1986.

[2] M. Broy, "Compositional Refinement of Interactive Systems", Report Number 89, Digital Systems Research Center, Palo Alto, CA, July, 1992.

[3] F. Gilham, R. A. Riemenschneider, and V. Stavridou, "Secure Interoperability of Secure Distributed Databases: An Architecture Verification Case Study", *FM '99, World Congress on Formal Methods,* Toulouse, France, September 20-24, 1999.

[4] S. Graf and H. Saïdi, "Construction of Abstract State Graphs Using PVS", *Proceedings of the 9th International Conference on Computer-Aided Verification, CAV '97,* Haifa, Israel, 1997.

[5] J. Herbert, B. Dutertre, R. A. Riemenschneider, and V. Stavridou, "A Formalization of Software Architecture", *FM '99, World Congress on Formal Methods*, Toulouse, France, September 20-24, 1999.

[6] M. R. Lowry and R. Duran, "Knowledge-Based Software Engineering". In A. Barr, P. R. Cohen, and E. A. Feigenbaum, *The Handbook of Artificial Intelligence, Volume IV*, Addison-Wesley, 1989.

[7] M. R. Lowry and R. D. McCartney, editors, *Automating Software Design*, AAAI Press/MIT Press, 1991.

[8] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide", *IEEE Transactions on Software Engineering*, vol. 21, no. 4, April, 1995.

[9] M. Moriconi, X. Qian, and R. A. Riemenschneider, “Correct Architecture Refinement”, *IEEE Transactions on Software Engineering*, vol. 21, no. 4, April, 1995.

[10] M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong, “Secure Software Architectures”, *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May, 1997.

[11] M. Moriconi and R. A. Riemenschneider, “Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies”, Technical Report SRI-CSL-97-01, Computer Science Laboratory, SRI International, Menlo Park, CA, March, 1997.

[12] J. Philipps and B. Rumpe, “Refinement of Information Flow Architectures”, *Proceedings of the First IEEE International Conference on Formal Engineering Methods, ICFEM ’97*, November, 1997.

[13] C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[14] R. A. Riemenschneider, “A Simplified Method for Establishing the Correctness of Architectural Refinements”, Working Paper DSA-97-02, Dependable System Archiecture Group, Computer Science Laboratory, SRI International, Menlo Park, CA, November, 1997. Available at `http://www.sdl.sri.com/papers/simplified/`.

[15] R. A. Riemenschneider, “Correct Transformation Rules for Incremental Development of Architecture Hierarchies”, Working Paper DSA-98-01, Dependable System Architecture Group, Computer Science Laboratory, SRI International, Menlo Park, CA, February, 1998. Available at `http://www.sdl.sri.com/papers/incremental/`.

[16] R. A. Riemenschneider. “Checking the Correctness of Architectural Transformation Steps via Proof-Carrying Architectures”. In P. Donahoe, *Software Architecture*, Kluwer, 1999.

[17] R. A. Riemenschneider and V. Stavridou. “The Role of Architecture Description Languages in Component-Based Development: The SRI Perspective”, *1999 International Workshop on Component-Based Software Engineering*, Los Angeles, CA, May 17-18, 1999.

[18] H. Saïdi and N. Shankar, “Abstract and Model-Check While You Prove”, *Proceedings of the 11th International Conference on Computer-Aided Verification, CAV ’99*, Trento, Italy, 1999.

[19] H. Saïdi, “Modular and Incremental Analysis of Concurrent Software Systems”, *Proceedings of the 14th IEEE International Conference on Automated Software Engineering, ASE ’99*, Cocoa Beach, FL, 1999.

[20] T. Saridakis and V. Issarny, “Developing Dependable Systems Using Software Architecture”in P. Donahoe, *Software Architecture*, Kluwer, 1999.